Linux 的高级路由和流量控制 HOWTO

中文版

Bert Hubert

Netherlabs BV

bert.hubert@netherlabs.nl

Gregory Maxwell (章节作者)

remco%virtu.nl

Remco van Mook (章节作者)

remco@virtu.nl

Martijn van Oosterhout (章节作者)

kleptog@cupid.suninternet.com

Paul B Schroeder (章节作者)

paulsch@us.ibm.com

Jasper Spaans (章节作者)

jasper@spaans.ds9a.nl

Pedro Larroy (章节作者)

piotr%omega.resa.ed

一个非常上手的关于 iproute2、流量整形和一点 netfilter 的指南。

译者序

可以说，翻译这篇文档的过程就是我重新学习 Linux 的过程。与原文的作者感受相似，当我根据这篇文档大致了解了 Linux 在 IP 方面的功能后，绝对是“ it *really* blew me away !”。才发现我以前一直是把 Linux 当成 UNIX 来用，Linux 本身很多精彩的功能都被我忽略了。

看来 Linux 在路由方面的设计的确非常独到。

但愿这篇文章的内容能够对您应用 Linux 有所帮助。

本文档的原作实际上还尚未完成，估计要等到 Linux 的 2.6 版内核发布之后才能最终定稿。但是我已经等不及了，非常希望尽快与各位共享这篇文档。如果这篇文档的原作完成，我会尽力追踪翻译。

这里是本 HOWTO 的正规出处。

由于本人的英语和语文都是业余水平，有两三处晦涩或者与技术无关的内容没有翻译，希望英语高手予以指点。如有词不达意甚至理解错误之处，非常渴望您能通过 Email 告知！谢谢！

2/15/2003 5:28 PM　译毕

JohnBull <deathumb@95700.net>

目录

第 1 章 贡献

本文档的成形得益于很多人的贡献，我希望能够回报他们。列出其中几个：

- Rusty Russell
- Alexey N. Kuznetsov
- 来自 Google 的一些好心人
- Casema Internet 的工作人员

第 2 章 简介

欢迎，亲爱的读者。

希望这篇文档能对你更好地理解 Linxs2.2/2.4 的路由有所帮助和启发。不被大多数使用者所知道的是，你所使用工具，其实能够完成相当规模工作。比如 route 和 ifconfig，实际上暗中调用了非常强大的 iproute 2 的底层基本功能。

我希望这个 HOWTO 能够象 Rusty Russell 的作品那样通俗易懂。

你可以随时给 HOWTO 工作组发电子邮件来找到我们。但是如果您的问题并不直接与这个 HOWTO 文档相关，请首先考虑发给邮件列表(参考相关章节)。我们可不是免费的帮助平台，但我们经常会在邮件列表上回答问题。

在钻研这个 HOWTO 之前，如果您想做的只是一点简单的流量整形，不妨直接去看看其它可能性这一章里面的 CBQ.init。

2.1. 除外责任与许可

这个文档依着对公众有利用价值的目的而发布，但**不提供任何担保**，即使是在**经销或者使用在特定场合**时的潜在担保。

简单地说，如果您的 STM-64 骨干网瘫痪，并向您尊敬的客户们散布黄色图片，对不起，那绝对不关我的事。

请随意复制并发布(出售或者赠送)本文档，格式不限。只是请求将纠正和/或注解转发给文档的维护者。

还希望如果你出版本 HOWTO 的硬拷贝，请给作者们发一份以备复习之用。☺

2.2. 预备知识

就像标题所暗示的，这是一个“高级”HOWTO。虽然它不是终极的航天科技，但还是要求一定的基础知识。

这里是一些可能对你有帮助的参考文献：

Rusty Russell 的 networking-concepts-HOWTO

非常精彩的介绍，解释了什么是网络以及一个网络如何与其它网络互联。

Linux Networking-HOWTO (以前叫做 Net-3 HOWTO)

好东西，虽然非常冗长。它讲授的内容就是你连接到 Internet 所需的的配置内容。应该在/usr/doc/HOWTO/NET3-4-HOWTO.txt 中，也可以在线阅读。

2.3. Linux 能为你做什么

一个小列表：

- 管制某台计算机的带宽
- 管制通向某台计算机的带宽
- 帮助你公平地共享带宽
- 保护你的网络不受 DoS 攻击
- 保护 Internet 不受到你的客户的攻击
- 把多台服务器虚拟成一台，进行负载均衡或者提高可用性
- 限制对你的计算机的访问
- 限制你的用户访问某些主机
- 基于用户账号(没错！)、MAC 地址、源 IP 地址、端口、服务类型、时间或者内容等条件进行路由。

现在，很多人都没有用到这些高级功能。这有很多原因。比如提供的文档过于冗长而且不容易上手，而且流量控制甚至根本就没有归档。

2.4. 内务声明

关于这个文档有些事情要指出。当我写完这个文档的绝大部分的时候，我真的不希望它永远就是那个样子。我是一个坚信开放源代码的人，所以我希望你能够给我发回反馈、更新、补丁等等。所以你应该尽可以告知我你的手稿或者指出一些哪怕是无关紧要的错误，不必犹豫。如果我的英语有些晦涩，请原谅那不是我的母语，尽可以给我建议。

如果你认为自己更有资格维护某个章节，或者认为自己可以写作并维护一个新的章节，请您一定不要客气。这个 HOWTO 的 SGML 可以通过 CVS 得到，我估计肯定有很多人还在为它出力。

作为请求援助，你会在文档中发现很多“ 求助 ”的字样。我们永远欢迎您的补丁！无论您在哪里发现 “ 求助 ”，都应该明白您正在踏入一个未知的领域。这并不是说在别的地方就没有错误，但您应该倍加小心。如果您确认了某些事情，请您一

定通知我们，以便我们能够把“求助”的标记去掉。

关于这个 HOWTO，I will take some liberties along the road. For example, I postulate a 10Mbit Internet connection, while I know full well that those are not very common.

2.5. 访问、CVS 和提交更新

本 HOWTO 的规范位置在这里。

我们现在向全球开放了匿名 CVS 访问。从各个角度来说这都是一件好事。你可以轻松地升级到本 HOWTO 的最新版本，而且提交补丁也不再成为问题。

另外的好处是，这可以让作者在源码上独立地继续工作。

```
$ export CVSROOT=:pserver:anon@outpost.ds9a.nl:/var/cvsroot
$ cvs login
CVS password: [enter 'cvs' (without 's)]
$ cvs co 2.4routing
cvs server: Updating 2.4routing
U 2.4routing/lartc.db
```

如果您做了修改并希望投稿，运行：

cvs -z3 diff –uBb

然后把输出用电子邮件发给<howto@ds9a.nl>，我们就可以很轻松地把它集成进去了。谢谢！请确认你修改的是.db 文件，其它文件都是通过它生成的。

提供了一个 Makefile 帮助您生成 postscript、dvi、pdf、html 和纯文本格式的文件。你可能需要安装 docbook、docboot-utils、ghostscript 和 tetex 等等支持软件才能生成各种格式的文本。

注意，不要更改 2.4routing.sgml！那里面有旧版本的 HOWTO。正确的文件是 lartc.db。

2.6. 邮件列表

作者已经开始收到关于这个 HOWTO 越来越多的邮件了。为了把大家的兴趣条理化，已经决定启动一个邮件列表，让大家在那里互相探讨有关高级路由和流量控制的话题。你可以在这里进行订阅。

需要指出的是，作者们对于列表中没有问及的问题不可能及时回答。我们愿意让列表的归档成为一个知识库。如果你有问题，请搜索归档，然后在 post 到邮件列表里。

2.7. 本文档的布局

我们几乎马上就要做一些有趣的实验，也就意味着最开始部分的基本概念解释并不完整或者不完善，请您不必管它，后面会一点点说清楚。

路由和包过滤是完全不同的概念。关于过滤的问题，Rusty 的文档说得很清楚，你可以在这里找到：

- Rusty 出色的不可靠指南

我们则将致力于 netfilter 与 iproute2 相结合后能做什么。

第 3 章 介绍 iproute2

3.1 为什么使用 iproute2?

现在，绝大多数 Linux 发行版和绝大多数 UNIX 都使用古老的 **arp**, **ifconfig** 和 **route** 命令。虽然这些工具能够工作，但它们在 Linux2.2 和更高版本的内核上显得有一些落伍。比如，现在 GRE 隧道已经成为了路由的一个主要概念，但却不能通过上述工具来配置。

使用了 iproute2，隧道的配置与其他部分完全集成了。

2.2 和更高版本的 Linux 内核包含了一个经过彻底重新设计的网络子系统。这些新的代码让 Linux 在操作系统的竞争中取得了功能和性能上的优势。实际上，Linux 新的路由、过滤和分类代码，从功能和性能上都不弱于现有的那些专业的路由器、防火墙和流量整形产品。

随着新的网络概念的提出，人们在现有操作系统的现有体系上修修补补来实现他们。这种固执的行为导致了网络代码中充斥着怪异的行为，这有点像人类的语言。过去，Linux 模仿了 SunOS 的许多处理方式，并不理想。

这个新的体系则有可能比以往任何一个版本的 Linux 都更善于清晰地进行功能表达。

3.2 iproute2 概览

Linux 有一个成熟的带宽供给系统，称为 Traffic Control（流量控制）。这个系统支持各种方式进行分类、排序、共享和限制出入流量。

我们将从 iproute2 各种可能性的一个简要概览开始。

3.3 先决条件

你应该确认已经安装了用户级配置工具。这个包的名字在 RedHat 和 Debian 中都叫作 " iproute "，也可以在这个地方找到：

ftp://ftp.inr.ac.ru/ip-routing/iproute2-2.2.4-now-ss??????.tar.gz

你也可以试试在这里找找最新版本。

iproute 的某些部分需要你打开一些特定的内核选项。应该指出的是，RedHat6.2 及其以前的所有发行版中所带的缺省内核都不带有流量控制所需要的绝大多数功能。

而 RedHat 7.2 在缺省情况下能满足所有要求。

另外，确认一下你的内核支持 netlink ，Iproute2 需要它.

3.4 浏览你的当前配置

这听上去确实让人惊喜：iproute2 已经配置好了！当前的 **ifconfig** 和 **route** 命令已经正在使用新的系统调用，但通常使用了缺省参数(真无聊)。

新的工具 **ip** 成为中心，我们会让它来显示我们的网卡配置。

3.4.1. 让 ip 显示我们的链路

```
[ahu@home ahu]$ ip link list
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1400 qdisc pfifo_fast qlen 100
    link/ether 48:54:e8:2a:47:16 brd ff:ff:ff:ff:ff:ff
4: eth1: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:e0:4c:39:24:78 brd ff:ff:ff:ff:ff:ff
3764: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP> mtu 1492 qdisc pfifo_fast qlen 10
    link/ppp
```

你的结果可能有所区别，但上述显示了我家里 NAT 路由器的情况。我将只解释输出中并非全部直接相关的部分。因为并不是所有部分都与我们的话题有关，所以我只会解释输出的一部分。

我们首先看到了 loopback 接口。 While your computer may function somewhat without one, I'd advise against it. MTU (最大传输单元)尺寸为 3924 字节，并且不应该参与队列。这是因为 loopback 接口完全是内核想象出来的、并不存在的接口。

现在我们跳过这个无关的接口，它应该并不实际存在于你的机器上。然后就是两个物理网络接口，一个接在我的 cable modem 上，另一个接到我家里的以太网端上。再下面，我们看见了一个 ppp0 接口。

应该指出，我们没有看到 IP 地址。iproute 切断了“链路”和“IP 地址”两个概念的直接联系。当使用 IP 别名的时候，IP 地址的概念显得更加不相关了。

尽管如此，还是显示出了标识以太网卡硬件的 MAC 地址。

3.4.2. 让 ip 显示我们的 IP 地址

```
[ahu@home ahu]$ ip address show
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop
```

```
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1400 qdisc pfifo_fast qlen 100
    link/ether 48:54:e8:2a:47:16 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/8 brd 10.255.255.255 scope global eth0
4: eth1: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:e0:4c:39:24:78 brd ff:ff:ff:ff:ff:ff
3764: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP> mtu 1492 qdisc pfifo_fast qlen 10
    link/ppp
    inet 212.64.94.251 peer 212.64.94.1/32 scope global ppp0
```

这里包含了更多信息。显示了我们所有的地址，以及这些地址属于哪些网卡。"inet"表示 Internet (IPv4)。还有很多其它的地址类型，但现在还没有涉及到。

让我们先就近看看 eth0。上面说它与 IP 地址 10.0.0.1/8 相关联。这是什么意思呢？"/8"表示 IP 地址表示网络地址的位数。因为一共是 32 个 bit，所以我们的这个网络有了 24 bit 的主机空间。 10.0.0.1 的开始 8bit 是 10.0.0.0,也就是我们的网络地址，我们的子网掩码是 255.0.0.0。

其它的 bit 直接连接在这个网卡上，所以 10.250.3.13 可以直接通过 eth0 联络到，就象 10.0.0.1 一样。

对于 ppp0，仍是相同的概念，虽然数字看上去有所不同。它的地址是 212.64.94.251，不带子网掩码。这意味着这是一个点到点的连接，而且除了 212.64.94.251 之外的地址是对端的。当然，还有很多信息。它还告诉我们这个链路的另一端只有一个地址：212.64.94.1。/32 意思是说没有表示网络的 bit。

掌握这些概念是绝对重要的。如果有问题，不妨先参考以下这个 HOWTO 文件开头曾经提到的那些文档。

你应该注意到了"qdisc"，它是基于对列规范的一个概念。它在后面会变得很重要。

3.4.3. 让 ip 显示路由

好的，现在我们已经知道如何找到 10.x.y.z 了，然后我们就可以到达 212.64.94.1。但这还不够，我们还得说明如何找到全世界。可以通过我们的 ppp 连接找到 Internet，212.64.94.1 愿意把我们的数据包发给全世界，并把回应的数据包传回给我们。

```
[ahu@home ahu]$ ip route show
212.64.94.1 dev ppp0  proto kernel  scope link  src 212.64.94.251
10.0.0.0/8 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 212.64.94.1 dev ppp0
```

字面的意思相当清楚。前 4 行的输出明确地说明了 **ip address show** 的意思，最后一行说明了世界的其它部分可以通过我们的缺省网关 212.64.94.1 找到。我们通过"via"这个词断定这是一个网关，我们要把数据包交给它。这就是我们要留心的问题

下面列出以前 **route** 命令的输出作为参考：

```
[ahu@home ahu]$ route -n
```

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
212.64.94.1     0.0.0.0         255.255.255.255 UH    0      0        0 ppp0
10.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
0.0.0.0         212.64.94.1     0.0.0.0         UG    0      0        0 ppp0
```

3.5. ARP

ARP 是由 RFC 826 所描述的“ 地址解析协议 ”。ARP 是网络上的计算机在居域网中用来解析另一台机器的硬件地址/位置的时候使用的。互联网上的机器一般都是通过机器名解析成 IP 地址来互相找到的。这就能够解决 foo.com 网络能够与 bar.net 网络通讯。但是，仅仅依靠 IP 地址，却无法得到一台计算机在一个网络中的物理位置。这时候就需要 ARP。

让我们举一个非常简单的例子。假定我有一个网络，里面有几台机器。其中的两台在我的子网上，一台叫 foo，IP 地址是 10.0.0.1，另一台叫 bar，IP 地址是 10.0.0.2。现在，foo 想 ping 一下 bar 看看是不是正常，但是呢，foo 只知道 bar 的 IP 地址，却并不知道 bar 的硬件(MAC)地址。所以 foo 在 ping bar 之前就会先发出 ARP 询问。这个 ARP 询问就像在喊：“Bar(10.0.0.2)!你在哪里(你的 MAC 地址是多少)？！”结果这个广播域中的每台机器都能听到 foo 的喊话，但是只有 bar(10.0.0.2)会回应。Bar 会直接给 foo 发送一个 ARP 回应，告诉它“ Foo (10.0.0.1)，我的 Mac 地址是 00:60:94:E9:08:12 ”。经过这种简单的交谈，机器就能够在局域网中定位它要通话的对象。Foo 会一直使用这个结果，直到它的 ARP 缓冲忘掉这个结果(在 Unix 系统上通常是 15 分钟之后)。

现在我们来看一看具体的工作过程。你可以这样察看你的 ARP 表（缓冲）：

```
[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud reachable
```

你可以看到，我的机器 espa041 (9.3.76.41) 知道如何找到 espa042 (9.3.76.42) 和 espagate (9.3.76.1)。现在让我们往缓冲中添加另一台机器。

```
[root@espa041 /home/paulsch/.gnome-desktop]# ping -c 1 espa043
PING espa043.austin.ibm.com (9.3.76.43) from 9.3.76.41 : 56(84) bytes of data.
64 bytes from 9.3.76.43: icmp_seq=0 ttl=255 time=0.9 ms

--- espa043.austin.ibm.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.9/0.9/0.9 ms

[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.43 dev eth0 lladdr 00:06:29:21:80:20 nud reachable
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud reachable
```

由于 espa041 试图联络 espa043，espa043 的硬件地址已经添加到 ARP 缓冲里了。所以直到 espa043 的记录失效以前(也就是两个机器间长时间没有通讯)，espa041 知道如何找到 espa043，也就不必频繁地进行 ARP 询问了。

现在让我们来删除 espa043 的 ARP 缓冲：

```
[root@espa041 /home/src/iputils]# ip neigh delete 9.3.76.43 dev eth0
[root@espa041 /home/src/iputils]# ip neigh show
9.3.76.43 dev eth0  nud failed
9.3.76.42 dev eth0 lladdr 00:60:08:3f:e9:f9 nud reachable
9.3.76.1 dev eth0 lladdr 00:06:29:21:73:c8 nud stale
```

现在 espa041 已经忘记了 espa043 的 MAC 地址,如果下次它要与 espa043 通讯，需要再次发送 ARP 询问。你在 espagate (9.3.76.1) 上也会发现以上输出已经变成了"stale"状态。这意味着 MAC 地址仍然是在册，但是接下来第一次通讯的时候需要确认一下。

第 4 章 规则——路由策略数据库

如果你有一个大规模的路由器，你可能不得不同时满足不同用户对于路由的不同需求。路由策略数据库可以帮助你通过多路由表技术来实现。

如果你想使用这个特性，请确认你的内核配置中带有 "IP: advanced router" 和 "IP: policy routing" 两项。

当内核需要做出路由选择时，它会找出应该参考哪一张路由表。除了 "ip" 命令之外，以前的 "route" 命令也能修改 main 和 local 表。

缺省规则：

```
[ahu@home ahu]$ ip rule list
0:
                                                                      f
rom all lookup local
32766:
                                                                      f
rom all lookup main
32767:
                                                                      f
rom all lookup default
```

上面列出了规则的优先顺序。我们看到，所有的规则都应用到了所有的包上 (“ from all ”)。我们前面已经看到了 "main" 表，就是“ **ip route ls** ”命令的输出，但是“ local ”和“ default ”是初次见到。

如果我们想做点有趣的事情，就可以生成一些指向不同路由表的规则，取代系统中的路由规则。

对于内核如何处理一个 IP 包匹配多个规则的精确意义，请参见 Alexey 关于 ip-cref 文档。

4.1. 简单的源策略路由

让我们再来一个真实的例子。我有两个 Cable Modem，连接到了一个 Linux 的 NAT (“ 伪装 ”) 路由器上。这里的室友们向我付费使用 Internet。假如我其中的一个室友因为只想访问 hotmail 而希望少付一些钱。对我来说这没有问题，他们肯定只能使用那个比较次的 Cable Modem。

那个比较快的 cable modem 的 IP 地址是 212.64.94.251， PPP 链路，对端 IP 是 212.64.94.1。而那个比较慢的 cable modem 的 IP 地址是 212.64.78.148，对端是 195.96.98.253。

local 表：

```
[ahu@home ahu]$ ip route list table local
```

```
broadcast 127.255.255.255 dev lo  proto kernel  scope link  src 127.0.0.1
local 10.0.0.1 dev eth0  proto kernel  scope host  src 10.0.0.1
broadcast 10.0.0.0 dev eth0  proto kernel  scope link  src 10.0.0.1
local 212.64.94.251 dev ppp0  proto kernel  scope host  src 212.64.94.251
broadcast 10.255.255.255 dev eth0  proto kernel  scope link  src 10.0.0.1
broadcast 127.0.0.0 dev lo  proto kernel  scope link  src 127.0.0.1
local 212.64.78.148 dev ppp2  proto kernel  scope host  src 212.64.78.148
local 127.0.0.1 dev lo  proto kernel  scope host  src 127.0.0.1
local 127.0.0.0/8 dev lo  proto kernel  scope host  src 127.0.0.1
```

有很多明显的事实，其实可能还需要进一步说明。好了，这样就行了。“ default ”表为空。

让我们看看 “ main ” 路由表：

```
[ahu@home ahu]$ ip route list table main
195.96.98.253 dev ppp2  proto kernel  scope link  src 212.64.78.148
212.64.94.1 dev ppp0  proto kernel  scope link  src 212.64.94.251
10.0.0.0/8 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 212.64.94.1 dev ppp0
```

我们现在为我们的朋友创建了一个叫做 “ John ” 的规则。其实我们完全可以使用纯数字表示规则，但是不方便。我们可以向 /etc/iproute2/rt_tables 文件中添加数字与名字的关联：

```
# echo 200 John >> /etc/iproute2/rt_tables
# ip rule add from 10.0.0.10 table John
# ip rule ls
0:
                                                                                  f
rom all lookup local
32765:
                                                                                  f
rom 10.0.0.10 lookup John
32766:
                                                                                  f
rom all lookup main
32767:
                                                                                  f
rom all lookup default
```

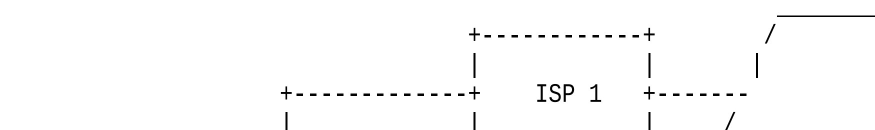
现在，剩下的事情就是为 John 的路由表创建路由项了。别忘了刷新路由缓存：

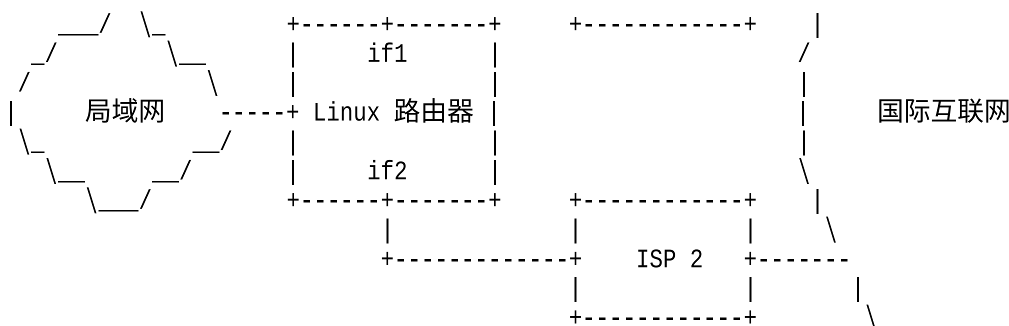
```
# ip route add default via 195.96.98.253 dev ppp2 table John
# ip route flush cache
```

这样就做好了。至于如何在 ip-up 阶段实现就留给读者自己去研究吧。

4.2. 多重上连 ISP 的路由

下图是很常见的配置，同一个局域网（甚至是同一台计算机）通过两个 ISP 连接到互联网上。





这种情况下通常会出现两个问题。

4.2.1. 流量分割

首先是如何保证：回应来自某一个 ISP 的数据包时，仍然使用相同的 ISP。

让我们先定义一些符号。 令第一块网卡(上图的 if1)的名字叫 **$IF1**，而第二块网卡叫做 **$IF2** 。然后设置 **$IF1** 的 IP 地址为 **$IP1**，**$IF2** 的 IP 地址为 **$IP2**。并且，令 ISP1 的网关地址为 **$P1**，ISP2 的网关地址为 **$P2**。最后，令**$P1** 的网络地址为 **$P1_NET** ，令**$P2** 的网络地址为 **$P2_NET**。

额外创建两个路由表， **T1** 和 **T2**。 加入到 /etc/iproute2/rt_tables 中。然后如下设置两个路由表中的路由：

```
ip route add $P1_NET dev $IF1 src $IP1 table T1
ip route add default via $P1 table T1
ip route add $P2_NET dev $IF2 src $IP2 table T2
ip route add default via $P2 table T2
```

没什么大不了的，不过是建立了通向该网关的一条路由，并使之成为默认网关，分别负责一个单独的上行流，并且为这两个 ISP 都作这样的配置。要指出的是，那条网络路由是必要条件，因为它能够让我们找到那个子网内的主机，也包括上述那台网关。

下一步，我们设置“ main ”路由表。把包通过网卡直接路由到与网卡相连的局域网上不失为一个好办法。要注意“ src ” 参数，他们能够保证选择正确的出口 IP 地址。

```
ip route add $P1_NET dev $IF1 src $IP1
ip route add $P2_NET dev $IF2 src $IP2
```

然后，设置你的缺省路由：

```
ip route add default via $P1
```

接着，设置路由规则。这实际上在选择用什么路由表进行路由。你需要确认当你从一个给定接口路由出数据包时，是否已经有了相应的源地址：你需要保证的就是如果你已经有了相应的源地址，就应该把数据包从相应的网卡路由出去：

```
ip rule add from $IP1 table T1
ip rule add from $IP2 table T2
```

以上命令保证了所有的回应数据都会从他们来的那块网卡原路返回。

现在，完成了非常基本的配置。这将对于所有运行在路由器上所有的进程起作用，实现 IP 伪装以后，对本地局域网也将起作用。如果不进行伪装，那么你要么拥有两个 ISP 的地址空间，要么你想对两个 ISP 中的一个进行伪装。无论哪种情况，你都要添加规则，基于发包的主机在局域网内的 IP 地址，选择从哪个 ISP 路由出去。

4.2.2. 负载均衡

第二个问题是如何对于通过两个 ISP 流出的数据进行负载均衡。如果你已经成功地实现了流量分割，这件事并不难。

与选择两个 ISP 中的一个作为缺省路由不同，这次是设置缺省路由为多路路由。在缺省内核中，这会均衡两个 ISP 的路由。象下面这样做(基于前面的流量分割实验):

```
ip route add default scope global nexthop via $P1 dev $IF1 weight 1 \
nexthop via $P2 dev $IF2 weight 1
```

这样就可以均衡两个 ISP 的路由。通过调整 " **weight** " 参数我们可以指定其中一个 ISP 的优先权高于另一个。

应该指出，由于均衡是基于路由进行的，而路由是经过缓冲的，所以这样的均衡并不是 100%精确。也就是说，对于一个经常访问的站点，总是会使用同一个 ISP。

进而，如果你对此不满意，你可能需要参考以下 Julian Anastasov 的内核补丁：

http://www.linuxvirtualserver.org/~julian/#routes

Julian 的路由补丁会弥补上述缺陷。

第 5 章 GRE 和其他隧道

Linux 有 3 种隧道。它们是：IP-in-IP 隧道、GRE 隧道和非内核隧道(如 PPTP)。

5.1. 关于隧道的几点注释

隧道可以用于实现很多非常不一般而有趣的功能。但如果你的配置有问题，却也会发生可怕的错误。除非你*确切地*知道你在做什么，否则不要把缺省路由指向一个隧道设备。而且，隧道会增加协议开销，因为它需要一个额外的 IP 包头。一般应该是每个包增加 20 个字节，所以如果一个网络的 MTU 是 1500 字节的话，使用隧道技术后，实际的 IP 包长度最长只能有 1480 字节了。这倒不是什么原则性的问题，但如果你想使用隧道技术构建一个比较大规模的网络的话，最好仔细研究一下关于 IP 包的分片和汇聚的知识。哦，还有，挖一个隧道最好的方法当然是同时从两头挖。

5.2. IP-in-IP 隧道

这种隧道在 Linux 上已经实现很长一段时间了。需要两个内核模块：ipip.o 和 new_tunnel.o。

比如说你有 3 个网络：内部网 A 和 B，中间网 C(比如说：Internet)。A 网络的情况：

网络地址

1

0.0.1.0
子网掩码

2

55.255.255.0
路由器

1

0.0.1.1

路由器在 C 网络上的地址是 172.16.17.18。

B 网络的情况：

网络地址

1

0.0.2.0
子网掩码

2

55.255.255.0
路由器

1

0.0.2.1

路由器在 C 网络上的 IP 地址是 172.19.20.21。

已知 C 网络已经连通，我们假定它会将所有的数据包从 A 传到 B，反之亦然。而且你可以随便使用 Internet。

这就是你要做的：

首先，确认模块是否加载：

```
insmod ipip.o
insmod new_tunnel.o
```

然后，在 A 网络的路由器上输入：

```
ifconfig tunl0 10.0.1.1 pointopoint 172.19.20.21
route add -net 10.0.2.0 netmask 255.255.255.0 dev tunl0
```

并且在 B 网络的路由器上输入：

```
ifconfig tunl0 10.0.2.1 pointopoint 172.16.17.18
route add -net 10.0.1.0 netmask 255.255.255.0 dev tunl0
```

如果你想中止隧道，输入：

```
ifconfig tunl0 down
```

简单之极！但是你不能通过 IP-in-IP 隧道转发广播或者 IPv6 数据包。你只是连接了两个一般情况下无法直接通讯的 IPv4 网络而已。至于兼容性，这部分代码已经有很长一段历史了，它的兼容性可以上溯到 1.3 版的内核。据我所知，Linux 的 IP-in-IP 隧道不能与其他操作系统或路由器互相通讯。它很简单，也很有效。需要它的时候尽管使用，否则就使用 GRE。

5.3. GRE 隧道

GRE 是最初由 CISCO 开发出来的隧道协议，能够做一些 IP-in-IP 隧道做不到的事情。比如，你可以使用 GRE 隧道传输多播数据包和 IPv6 数据包。在 Linux 下，你需要 ip_gre.o 模块。

5.3.1. IPv4 隧道

让我们先来做一做 IPv4 隧道：

比如说你有 3 个网络：内部网 A 和 B，中间网 C(比如说：Internet)。A 网络的情况：

网络地址

1

0.0.1.0

子网掩码

2

55.255.255.0

路由器

1

0.0.1.1

路由器在 C 网络上的地址是 172.16.17.18。我们称之为 neta。

B 网络的情况：

```
网络地址
                                                                                          1
0.0.2.0
子网掩码
                                                                                          2
55.255.255.0
路由器
                                                                                          1
0.0.2.1
```

路由器在 C 网络上的 IP 地址是 172.19.20.21。我们称之为 netb。

已知 C 网络已经连通，我们假定它会将所有的数据包从 A 传到 B，反之亦然。至于原因，我们不考虑。

在 A 网络的路由器上，输入：

```
ip tunnel add netb mode gre remote 172.19.20.21 local 172.16.17.18 ttl 255
ip link set netb up
ip addr add 10.0.1.1 dev netb
ip route add 10.0.2.0/24 dev netb
```

让我们稍微讨论一下。第 1 行，我们添加了一个隧道设备，并且称之为 netb(为了能够表示出这个隧道通向哪里)。并且表示要使用 GRE 协议 (mode gre),对端地址是 172.19.20.21(另一端的路由器)，我们的隧道数据包发源于 172.16.17.18(以便当你的路由器在 C 网络中拥有多个地址的时候，你可以指定哪一个应用于隧道)并且包的 TTL 字段应设置为 255(ttl 255)。

第 2 行，启用该隧道。

第 3 行，我们给这个新生的网卡配置了一个 IP：10.0.1.1。对于小网络来说足够了，但如果你网络中的隧道多得象无证运营的小煤窑一样，你可能就要考虑给你的隧道规划一个单独的 IP 地址范围(在本例中，你可以使用 10.0.3.0)。

第 4 行，我们为 B 网络设置了一条路由。注意子网掩码的另一种表示方法。如果你不熟悉这种表示，我就来解释一下：你把你的子网掩码写成二进制形式，数数里面由多少个 1。如果你连这个也不会做，不妨就简单地记住：255.0.0.0 就是 /8，255.255.0.0 就是 /16， 255.255.255.0 就是 /24。

让我们再看看 B 网络的路由器。

```
ip tunnel add neta mode gre remote 172.16.17.18 local 172.19.20.21 ttl 255
ip link set neta up
ip addr add 10.0.2.1 dev neta
ip route add 10.0.1.0/24 dev neta
```

如果你想从 A 路由器中停止隧道，输入：

```
ip link set netb down
ip tunnel del netb
```

当然，你可以把 netb 换成 neta，在 B 路由器上操作。

5.3.2. IPv6 隧道

关于 IPv6 地址，请参看第 6 章第 1 节。

这就开始吧。

我们假设你有如下的 IPv6 网络，你想把它连接到 6bone 或者一个朋友那里。

```
Network 3ffe:406:5:1:5:a:2:1/96
```

你的 IPv4 地址是 172.16.17.18，6bone 路由器的 IPv4 地址是 172.22.23.24。

```
ip tunnel add sixbone mode sit remote 172.22.23.24 local 172.16.17.18 ttl 255
ip link set sixbone up
ip addr add 3ffe:406:5:1:5:a:2:1/96 dev sixbone
ip route add 3ffe::/15 dev sixbone
```

让我们来讨论一下。我们创建了一个叫做 sixbone 的隧道设备。我们设置它的模式是 sit(也就是在 IPv4 隧道中使用 IPv6)并且告诉它对端(remote)和本端 (local)在哪里。TTL 设置为最大，255。接着，我们激活了这个设备(up)。然后，我们添加了我们自己的网络地址，并添加了一条通过隧道去往 3ffe::/15 (现在全部属于 6bone)的路由。

GRE 隧道是现在最受欢迎的隧道技术。它也广泛地应用于 Linux 世界之外并成为一个标准，是个好东西。

5.4. 用户级隧道

在内核之外，还有很多实现隧道的方法，最闻名的当然要数 PPP 和 PPTP，但实际上还有很多(有些是专有的，有些是安全的，有些甚至根本不用 IP)，但那远远超出了本 HOWTO 所涉及的范围。

第 6 章 用 Cisco 和 6bone 实现 IPv6

Marco Davids marco@sara.nl 著

NOTE to maintainer:

As far as I am concerned, this IPv6-IPv4 tunneling is not per definition GRE tunneling. You could tunnel IPv6 over IPv4 by means of GRE tunnel devices (GRE tunnels ANY to IPv4), but the device used here ("sit") only tunnels IPv6 over IPv4 and is therefore something different.

6.1. IPv6 隧道

这是 Linux 隧道能力的另一个应用。这在 IPv6 的早期实现中非常流行。下面动手试验的例子当然不是实现 IPv6 隧道的唯一方法。然而，它却是在 Linux 与支持 IPv6 的 CISCO 路由器之间搭建隧道的常用方法，经验证明多数人都是照这样做的。八成也适合于你☺。

简单谈谈 IPv6 地址：

相对于 IPv4 地址而言，IPv6 地址非常大，有 128bit 而不是 32bit。这让我们得到了我们需要的东西——非常非常多的 IP 地址。确切地说，有 340,282,266,920,938,463,463,374,607,431,768,211,465 个。同时，IPv6(或者叫 Ipng，下一代 IP)还能让 Internet 上的骨干路由器的路由表变得更小、设备的配置更简单、IP 层的安全性更好以及更好地支持 QoS。

例如: 2002:836b:9820:0000:0000:0000:836b:9886

写下一个 IPv6 地址确实是件麻烦事。所以我们可以使用如下规则来进行简化 ：

- 数字打头的零不要写，就像 IPv4 一样。
- 每 16bit 或者两个字节之间使用冒号分隔。
- 当出现很多连续的零时可简写成 “ :: ”。在一个地址中只能使用一次。

例如：地址 2002:836b:9820:0000:0000:0000:836b:9886 可以写成：2002:836b:9820::836b:9886，看上去更简单些。

另一个例子：地址 3ffe:0000:0000:0000:0000:0020:34A1:F32C 可以写成 3ffe::20:34A1:F32C，要短得多。

IPv6 将可能取代现有的 IPv4。因为它采用了相对更新的技术，所以现在还没有全球范围的 IPv6 网络。为了能够平滑地过渡，引入了 6bone 计划。

IPv6 网络中的站点通过现有的 IPv4 体系互联，把 IPv6 数据包封装在 IPv4 数据包中进行传输。

这就是为什么引入隧道机制的原因。

为了能够使用 IPv6，我们需要一个能够支持它的内核。现在有很多文档都很好地说明了这个问题。不外乎以下几步：

- 找到一个新版的 Linux 发行版，要有合适的 glibc 库。
- 找到一份最新的内核源代码。

都准备好了以后，就可以继续编译一个带 IPv6 支持的内核了：

- cd /usr/src/linux
- make menuconfig
- 选择 " Networking Options "
- 选择 " The IPv6 protocol "、" IPv6: enable EUI-64 token format ", " IPv6: disable provider based addresses "

提示：不要编译成内核模块，那样经常会出问题。换句话说，就是把 IPv6 内置入内核。

然后你就可以象往常一样保存配置并编译内核了。

提示：在编译之前，可以修改一下 Makefile，把 EXTRAVERSION = -x 变成 EXTRAVERSION = -x-IPv6

有很多文档都很好地说明了如何编译并安装一个内核，我们这篇文档不是讨论这个问题的。如果你在这个过程中出现了问题，请参阅合适的资料。你可以先看看 /usr/src/linux/README。

当你完成之后，用新的内核重启系统，你可以输入 " /sbin/ifconfig -a " 看看有没有新的 " sit0-device " 设备。SIT 的意思是 " 简单 Internet 过渡 " (Simple Internet Transition)。如果到这里没有问题，你就可以奖励自己了，你已经向着下一代 IP 网络迈进了一大步。

现在继续下一步。你需要把你的主机，或甚至整个局域网连接到另外一个 IPv6 网络上。这个网络很可能是" 6bone "，它就是为了这个特定的目的而专门设立的。

让我们假定你有如下 IPv6 网络: 3ffe:604:6:8::/64，并且希望连接到 6bone，或者其他地方。请注意，/64 这个子网声明的意义与 IPv4 相同。

你的 IPv4 地址是 145.100.24.181，6bone 的路由器的 IPv4 地址是 145.100.1.5。

```
# ip tunnel add sixbone mode sit remote 145.100.1.5 [local 145.100.24.181 ttl 255]
# ip link set sixbone up
# ip addr add 3FFE:604:6:7::2/126 dev sixbone
# ip route add 3ffe::0/16 dev sixbone
```

让我们讨论一下。第 1 行，我们创建了一个叫做 sixbone 的隧道。设置为 sit (让 IPv4 隧道承载 IPv6 数据包)模式，并设置对端与本端 IP。TTL 设为最大——255。

下一步，我们激活(up)了这个设备。然后添加我们自己的网络地址，并设置利用隧道通往 3ffe::/15 (which is currently all of 6bone) 的路由。如果你运行这个的这台机器是你的 IPv6 网关，就得考虑运行下面的命令：

```
# echo 1 >/proc/sys/net/ipv6/conf/all/forwarding
# /usr/local/sbin/radvd
```

下面的一行，radvd 是一个类似于 zebra 的路由公告守护程序，用来支持 IPv6 的自动配置特性。如果感兴趣的话就用你最喜欢的搜索引擎找一找。你可以检查一下：

```
# /sbin/ip -f inet6 addr
```

如果你的 Linux 网关支持 IPv6 且运行了 radvd，在局域网上启动后，你就可以享受 IPv6 的自动配置特性了：

```
# /sbin/ip -f inet6 addr
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue inet6 ::1/128 scope host

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
inet6 3ffe:604:6:8:5054:4cff:fe01:e3d6/64 scope global dynamic
valid_lft forever preferred_lft 604646sec inet6 fe80::5054:4cff:fe01:e3d6/10
scope link
```

你可以继续进行了，为 IPv6 配置你的 bind。与 A 记录等价的，支持 IPv6 的记录类型是“ AAAA ”。与 in-addr.arpa 等价的是“ ip6.int ”。这方面可以找到很多信息。

支持 IPv6 的应用系统曾在增加，包括 ssh、telnet、inetd、Mozilla 浏览器、Apache WEB 浏览器……。但那些都不是这个路由文档所应该涉及的。

作为 Cisco 系统，应该这样配置：

```
!
interface Tunnel1
description IPv6 tunnel
no ip address
no ip directed-broadcast
ipv6 address 3FFE:604:6:7::1/126
tunnel source Serial0
tunnel destination 145.100.24.181
tunnel mode ipv6ip
!
ipv6 route 3FFE:604:6:8::/64 Tunnel1
```

但如果你没有 Cisco 作为 disposal，试试 Internet 上的众多 IPv6 隧道提供者之一。他们愿意在他们的 Cisco 设备上为你额外创建一个隧道。大部分是友好的 WEB 界面。用你常用的搜索引擎搜索一下“ ipv6 tunnel broker ”。

第 7 章 IPSEC：Internet 上安全的 IP

现在，在 Linux 上有两种 IPSEC 可用。对于 2.2 和 2.4，第一个比较正规的实现有 FreeS/WAN。他们有一个官方站点和一个经常维护的非官方站点。出于多种原因，FreeS/WAN 历来没有被合并到内核的主线上。最常提到的原因就是政治问题——违反了美国的密码产品扩散条例。所以它不会被集成到 Linux 内核中。

另外，很多合作伙伴表明了他们对代码质量的忧虑。关于如何设置 FreeS/WAN，有很多文档可以参考。

Linux 2.5.47 版内核里有一个内置的 IPSEC 实现，是由 Alexey Kuznetsov 和 Dave Miller 在 USAGI IPv6 小组的启发下写成的。经过这次合并，James Morris 的 CrypoAPI 也成了内核的一部分——它能够真正地加密。

本 HOWTO 文档仅收录 2.5 以上版本内核的 IPSEC。Linux 2.4 内核的用户推荐使用 FreeS/WAN。但是要注意，它的配置方法与内置 IPSEC 不同。

2.5.49 版内核的 IPSEC 不再需要任何补丁。

- 我在这里收集了 Alexey 或 Dave Miller 发布的一些补丁。对于 2.5.48 版的内核，在报告 BUG 之前请确认已经打上了那些补丁！(迄今还没有 2.5.49 这方面的补丁)。一些简单的用户级工具可以在这里 (编译好的可执行文件和手册)找到。编译这些用户级工具需要修改 Makefiles 指向你的 2.5.x 内核。这种情况可望很快解决。

  编译你的内核的时候，要确认已经打开 CryptoAPI 中的“ PF_KEY ”、“ AH ”、“ ESP ” 以及其他所有选项！netfilter 中的 TCP_MSS 方法现在不能用，请关掉。

- 本章的作者对于 IPSEC 完全是外行（nitwit）！如果你发现了错误，请 email 通知 bert hubert <ahu@ds9a.nl>。

首先，我们展示一下如何在两个主机之间手动设置安全通讯。其间的大部分工作都可以自动完成，但是为了了解细节我们仍然用手动完成。

如果你仅仅对自动密钥管理感兴趣，请跳过下面一节。但是要知道，了解手动密要管理是很有用的。

7.1. 从手动密钥管理开始

IPSEC 是一个复杂的主题。有很多在线信息可以查阅，这个 HOWTO 将集中讲解如何设置、运行并解释一些基本原理。

- 很多 iptables 配置会丢弃 IPSEC 数据包！要想让 IPSEC 通过，运行

```
iptables -A xxx -p 50 -j ACCEPT
iptables -A xxx -p 51 -j ACCEPT
```

IPSEC 提供了一个安全的 IP 协议版本。所谓的“安全”意味着两件事情：加密与验证。如果认为安全仅仅就是加密哪就太天真了，很容易看出来那是不够的——你的通讯过程或许是加密的，但是你如何保证与你通讯的对端就是你期望中的人呢？

IPSEC 使用 ESP('Encapsulated Security Payload',安全载荷封装) 来支持加密，使用 AH(Authentication Header，头部验证)来支持对端验证。你可以同时使用二者，也可以使用二者之一。

ESP 和 AH 都依靠 SA(security associations,安全联盟)来工作。一个 SA 由一个源、一个目的和一个说明组成。一个验证 SA 看上去应该是：

```
add 10.0.0.11 10.0.0.216 ah 15700 -A hmac-md5 "1234567890123456";
```

意思是："从 10.0.0.11 到 10.0.0.216 的数据包需要 AH，使用 HMAC-MD5 签名，密码是 1234567890123456"。这个说明的 SPI(Security Parameter Index，安全参数索引)号码是'15700'，细节后面再谈。有意思的是一个会话的两端都使用相同的 SA——它们是全等的，而不是镜像对称的。要注意的是，SA 并没有自动翻转规则——这个 SA 仅仅描述了从 10.0.0.11 到 10.0.0.216 的认证。如果需要双向安全，需要 2 条 SA。

一个简单地 ESP SA:

```
add 10.0.0.11 10.0.0.216 esp 15701 -E 3des-cbc "123456789012123456789012";
```

意思是："从 10.0.0.11 到 10.0.0.216 的数据包需要 ESP，使用 3des-cbc 加密算法，密码是 123456789012123456789012"。SPI 号码是'15701'。

到此，我们看到 SA 描述了所有的说明，但它并没有描述应该什么时候用到这些策略。实际上，可以有很多完全相同的 SA，除了它们之间的 SPI 不同。顺便提一句，SPI 的意思是 Security Parameter Index(安全参数索引)。为了进行加密操作，我们需要声明一个策略。这个策略可以包含诸如"如果可能的话使用 ipsec"或者"除非我们有 ipsec 否则丢弃数据包"这样的东西。

最简单的典型 SP(Security Policy，安全策略)应该是这样：

```
spdadd 10.0.0.216 10.0.0.11 any -P out ipsec \
esp/transport//require \
ah/transport//require;
```

如果在 10.0.0.216 上输入这个，就意味着凡是去往 10.0.0.11 的数据包必须经过加密并附带 AH 头验证。要注意，它并没有指出使用哪一个 SA，那是留给内核来

决定的。

换句话说，SP 描述了我们需要**什么**；而 SA 描述了它应该**如何**实现。

发出的数据包打上 SA 的 API 标记，以便本端内核用来加密和鉴定、对端内核用来解密和验证。

接下来是一个非常简单的配置，用来从 10.0.0.216 到 10.0.0.11 使用加密和验证进行发包。注意。相反方向的发包仍然是明码传输的，没有进行相应的配置。

在 10.0.0.216 主机上：

```
#!/sbin/setkey -f
add 10.0.0.216 10.0.0.11 ah 24500 -A hmac-md5 "1234567890123456";
add 10.0.0.216 10.0.0.11 esp 24501 -E 3des-cbc "123456789012123456789012";

spdadd 10.0.0.216 10.0.0.11 any -P out ipsec
   esp/transport//require
   ah/transport//require;
```

在 10.0.0.11 机器上，SA 是一样的，没有 SP：

```
#!/sbin/setkey -f
add 10.0.0.216 10.0.0.11 ah 24500 -A hmac-md5 "1234567890123456";
add 10.0.0.216 10.0.0.11 esp 24501 -E 3des-cbc "123456789012123456789012";
```

经过上面的设置(如果你在/sbin 下安装了“ setkey ”命令，以上脚本是可以执行的)，从 10.0.0.216 上面 “ ping 10.0.0.11 ” 的过程用 tcpdump 应该截获如下内容：

```
22:37:52 10.0.0.216 > 10.0.0.11: AH(spi=0x00005fb4,seq=0xa): ESP(spi=0x00005fb5,seq=0xa) (DF)
22:37:52 10.0.0.11 > 10.0.0.216: icmp: echo reply
```

注意，可看出从 10.0.0.11 返回的包的确是明码传输。Ping 的进一步细节用 tcpdump 当然看不到，但是它还是显示了用来告诉 10.0.0.11 如何进行解密和验证的参数——AH 和 ESP 的 SPI 值。

还有几件事情必须提及。上面给出的配置在很多 IPSEC 的配置范例中都有引用，但确实是很危险的。问题就在于上述配置包含了 10.0.0.216 应该如何处理发往 10.0.0.11 的包，和 10.0.0.1 如何解释那些包，但是却**没有**指出 10.0.0.11 应当丢弃来自 10.0.0.216 的未进行加密及验证的包！

任何人都可以插入完全未加密的欺骗包，而 10.0.0.11 会不假思索地接受。为了弥补上述漏洞我们必须在 10.0.0.11 上增加一个针对进入数据包的 SP，如下：

```
#!/sbin/setkey -f
spdadd 10.0.0.216 10.0.0.11 any -P IN ipsec
   esp/transport//require
   ah/transport//require;
```

这就指明了 10.0.0.11 收到来自 10.0.0.216 的包的时候需要正确的 ESP 和 AH 处理。

现在，我们完成这个配置，我们当然也希望回去的数据包也进行加密和头验证。10.0.0.216 上完整的配置应该是：

```
#!/sbin/setkey -f
flush;
spdflush;

# AH
add 10.0.0.11 10.0.0.216 ah 15700 -A hmac-md5 "1234567890123456";
add 10.0.0.216 10.0.0.11 ah 24500 -A hmac-md5 "1234567890123456";

# ESP
add 10.0.0.11 10.0.0.216 esp 15701 -E 3des-cbc "123456789012123456789012";
add 10.0.0.216 10.0.0.11 esp 24501 -E 3des-cbc "123456789012123456789012";

spdadd 10.0.0.216 10.0.0.11 any -P out ipsec
           esp/transport//require
           ah/transport//require;

spdadd 10.0.0.11 10.0.0.216 any -P in ipsec
           esp/transport//require
           ah/transport//require;
```

10.0.0.11 上：

```
#!/sbin/setkey -f
flush;
spdflush;

# AH
add 10.0.0.11 10.0.0.216 ah 15700 -A hmac-md5 "1234567890123456";
add 10.0.0.216 10.0.0.11 ah 24500 -A hmac-md5 "1234567890123456";

# ESP
add 10.0.0.11 10.0.0.216 esp 15701 -E 3des-cbc "123456789012123456789012";
add 10.0.0.216 10.0.0.11 esp 24501 -E 3des-cbc "123456789012123456789012";


spdadd 10.0.0.11 10.0.0.216 any -P out ipsec
           esp/transport//require
           ah/transport//require;

spdadd 10.0.0.216 10.0.0.11 any -P in ipsec
           esp/transport//require
           ah/transport//require;
```

注意，本例中通信双方的加密密钥是一样的。这在实际中是不会出现的。

为了检测一下我们刚才的配置，运行一下：

```
setkey –D
```

就会显示出 SA，或者用

```
setkey –DP
```

显示出 SP。

7.2. 自动密钥管理

在上一节中，使用了简单共享的密码进行加密。换句话说，为了保密，我们必须通过一个安全的通道把加密配置传给对方。如果我们使用 telnet 配置对端的机器，任何一个第三方都可能知道我们的共享密码，那么设置就是不安全的。

而且，因为密码是共享的，所以它就不成为真正意义的密码。就算对端不能用它做什么，但我们需要为每一个要进行 IPSEC 通讯的对方都设置互补相同的密码。这需要生成大量的密钥，如果有 10 个成员需要通讯，就至少需要 50 个不同的密码。

除了对称密钥的问题之外，还有解决密钥轮转的问题。如果第三方搜集到足够的数据包，就有可能反向计算出密钥。这可以通过每隔一段时间换用一个新密钥来解决，但是这必须自动完成。

另一个问题是，如上所述的手动密钥管理，我们必须精确地指定算法和密钥长度，与对端的协调也是个大问题。我们渴望能够用一种比较宽松的方式来描述密钥策略，比如说：“我们可以用 3DES 或者 Blowfish 算法，密钥长度至少是多少多少位”。

为了满足这些要求，IPSEC 提供了 IKE(Internet Key Exchange，Internet 密钥交换)来自动随机生成密钥，并使用协商好的非对称加密算法进行密钥交换。

Linux 2.5 的 IPSEC 实现利用 KAME 的“racoon”IKE 守护程序来进行。截止到 11 月 9 日，在 Alexey 的 iptools 发布包中的 racoon 是可以编译的，但是需要从两个文件中删除#include <net/route.h>。你也可以下载我提供的编译好的版本。

- IKE 需要使用 UDP 的 500 端口，确认你的 iptables 不会挡住数据包。

7.2.1. 理论

象前面所解释的自动密钥管理会为我们做很多事情。特别地，它会自动地动态生成 SA。不象大家所以为的那样，它并不会为我们设置 SP。

所以，如果想使用 IKE，需要设置好 SP，但不要设置任何 SA。内核如果发现有一个 IPSEC 的 SP，但不存在任何相应的 SA，就会通知 IKE 守护程序，让它去协商。

重申，一个 SP 决定了我们需要**什么**；而 SA 决定了我们**如何**得到它。使用自动密钥管理就可以让我们只关心需要什么就够了。

7.2.2. 举例

Kame 的 racoon 有非常多的选项，其中绝大部分都已经配置好了缺省值，所以我们不用修改它们。象上面描述的，管理员需要定义一个 SP 而不配置 SA，留给 IKE 守护程序去协商。

在这个例子中，仍然是 10.0.0.11 和 10.0.0.216 之间需要配置安全通讯，但这次要

借助于 racoon。为了简单起见，这个配置使用预先共享的密钥（又是可怕的共享密钥）。X.509 证书的问题单独讨论，参见后面的 7.2.3。

我们尽量保持缺省配置，两台机器是一样的：

```
path pre_shared_key "/usr/local/etc/racoon/psk.txt";
remote anonymous
{
    exchange_mode aggressive,main;
    doi ipsec_doi;
    situation identity_only;
    my_identifier address;
    lifetime time 2 min;   # sec,min,hour
    initial_contact on;
    proposal_check obey;                                                          #
    obey, strict or claim
    proposal {
        encryption_algorithm 3des;
        hash_algorithm sha1;
        authentication_method pre_shared_key;
        dh_group 2 ;
    }
}

sainfo anonymous
{
 pfs_group 1;
 lifetime time 2 min;
 encryption_algorithm 3des ;
 authentication_algorithm hmac_sha1;
  compression_algorithm deflate ;
}
```

有很多设置，我认为仍然有很多可以去掉而更接近缺省配置。很少有值得注意的事情。我们已经配置了两个匿名配置支持所有的对端机器，让将来的配置简单些。这里没有必要为每台机器各写一个段落，除非真的必要。

此外，我们还设置了我们基于我们的 IP 地址来识别我们自己('my_identifier address')，并声明我们可以进行 3DES、sha1，并且我们将使用预先共享的密钥，写在 psk.txt 中。

在 psk.txt 中，我们设置两个条目，两台机器上都不一样。

在 10.0.0.11 上：

```
10.0.0.216
password2
```

在 10.0.0.216 上：

```
10.0.0.11
password2
```

确认这些文件必须为 root 所有、属性是 0600，否则 racoon 将不信任其内容。两个机器上的这个文件是镜像对称的。

现在，我们就剩下设置 SP 了，这比较简单。

在 10.0.0.216 上：

```
#!/sbin/setkey -f
flush;
spdflush;

spdadd 10.0.0.216 10.0.0.11 any -P out ipsec
esp/transport//require;

spdadd 10.0.0.11 10.0.0.216 any -P in ipsec
esp/transport//require;
```

在 10.0.0.11 上：

```
#!/sbin/setkey -f
flush;
spdflush;

spdadd 10.0.0.11 10.0.0.216 any -P out ipsec
esp/transport//require;

spdadd 10.0.0.216 10.0.0.11 any -P in ipsec
esp/transport//require;
```

请注意这些 SP 的镜像对称规律。

我们可以启动 racoon 了！一旦启动，当我们试图从 10.0.0.11 到 10.0.0.216 进行 telnet 的时候，racoon 就会开始协商：

```
12:18:44: INFO: isakmp.c:1689:isakmp_post_acquire(): IPsec-SA
  request for 10.0.0.11 queued due to no phase1 found.
12:18:44: INFO: isakmp.c:794:isakmp_ph1begin_i(): initiate new
  phase 1 negotiation: 10.0.0.216[500]<=>10.0.0.11[500]
12:18:44: INFO: isakmp.c:799:isakmp_ph1begin_i(): begin Aggressive mode.
12:18:44: INFO: vendorid.c:128:check_vendorid(): received Vendor ID:
  KAME/racoon
12:18:44: NOTIFY: oakley.c:2037:oakley_skeyid(): couldn't find
  the proper pskey, try to get one by the peer's address.
12:18:44: INFO: isakmp.c:2417:log_ph1established(): ISAKMP-SA
  established 10.0.0.216[500]-10.0.0.11[500] spi:044d25dede78a4d1:ff01e5b4804f0680
12:18:45: INFO: isakmp.c:938:isakmp_ph2begin_i(): initiate new phase 2
  negotiation: 10.0.0.216[0]<=>10.0.0.11[0]
12:18:45: INFO: pfkey.c:1106:pk_recvupdate(): IPsec-SA established:
  ESP/Transport 10.0.0.11->10.0.0.216 spi=44556347(0x2a7e03b)
12:18:45: INFO: pfkey.c:1318:pk_recvadd(): IPsec-SA established:
  ESP/Transport 10.0.0.216->10.0.0.11 spi=15863890(0xf21052)
```

如果我们现在运行 `setkey –D` 列出 SA，就会发现已经存在了：

```
10.0.0.216 10.0.0.11
esp mode=transport spi=224162611(0x0d5c7333) reqid=0(0x00000000)
E: 3des-cbc  5d421c1b d33b2a9f 4e9055e3 857db9fc 211d9c95 ebaead04
A: hmac-sha1  c5537d66 f3c5d869 bd736ae2 08d22133 27f7aa99
seq=0x00000000 replay=4 flags=0x00000000 state=mature
created: Nov 11 12:28:45 2002
current: Nov 11 12:29:16 2002
diff: 31(s)
hard: 600(s)
soft: 480(s)
last: Nov 11 12:29:12 2002
hard: 0(s)
soft: 0(s)
current: 304(bytes)
hard: 0(bytes)
soft: 0(bytes)
```

```
allocated: 3
hard: 0
soft: 0

sadb_seq=1 pid=17112 refcnt=0
10.0.0.11 10.0.0.216
esp mode=transport spi=165123736(0x09d79698) reqid=0(0x00000000)
E: 3des-cbc  d7af8466 acd4f14c 872c5443 ec45a719 d4b3fde1 8d239d6a
A: hmac-sha1  41ccc388 4568ac49 19e4e024 628e240c 141ffe2f
seq=0x00000000 replay=4 flags=0x00000000 state=mature
created: Nov 11 12:28:45 2002
current: Nov 11 12:29:16 2002
diff: 31(s)
hard: 600(s)
soft: 480(s)
last:
hard: 0(s)
soft: 0(s)
current: 231(bytes)
hard: 0(bytes)
soft: 0(bytes)
allocated: 2
hard: 0
soft: 0
sadb_seq=0 pid=17112 refcnt=0
```

我们的 SP 是如下配置的：

```
10.0.0.11[any] 10.0.0.216[any] tcp
in ipsec
esp/transport//require
created:Nov 11 12:28:28 2002 lastused:Nov 11 12:29:12 2002
lifetime:0(s) validtime:0(s)
spid=3616 seq=5 pid=17134
refcnt=3
10.0.0.216[any] 10.0.0.11[any] tcp
out ipsec
esp/transport//require
created:Nov 11 12:28:28 2002 lastused:Nov 11 12:28:44 2002
lifetime:0(s) validtime:0(s)
spid=3609 seq=4 pid=17134
refcnt=3
```

7.2.2.1. 问题和常见的疏忽

如果不工作，检查一下所有的配置文件是不是为 root 所有，而且只有 root 才能读取。如想前台启动 racoon，就加上“ -F ”参数。如想强制它读取某一个配置文件来取代缺省配置文件，使用参数“ -f ”。如想看到超级详细的细节，往 racoon.conf 中加入“ log debug; ”一行。

7.2.3. 使用 X.509 证书进行自动密钥管理

如前所述，之所以共享密码很困难，是因为它们一旦共享，就不再成为真正意义的密码。幸运的是，我们仍可以用非对称加密技术来解决这个问题。

如果 IPSEC 的每个参与者都生成一对公钥和私钥，就可以让双方公开它们的公

钥并设置策略，从而建立安全连接。

虽然需要一些计算，但生成密钥还是相对比较简单的。以下都是基于 openssl 工具实现的。

7.2.3.1. 为你的主机生成一个 X.509 证书

OpenSSL 搭好了很多基础结构，以便我们能够使用经过或者没有经过 CA 签署的密钥。现在，我们就围绕这些基础结构，并练习一下使用著名的 Snake Oil 安全，而不是使用 CA。

首先，我们为主机 laptop 发起一个 " 证书请求 "：

```
$ openssl req -new -nodes -newkey rsa:1024 -sha1 -keyform PEM -keyout \
  laptop.private -outform PEM -out request.pem
```

这是可能问我们的问题：

```
Country Name (2 letter code) [AU]:NL
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:Delft
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Linux Advanced
Routing & Traffic Control
Organizational Unit Name (eg, section) []:laptop
Common Name (eg, YOUR name) []:bert hubert
Email Address []:ahu@ds9a.nl

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

请你根据自己的实际情况完整填写。你可以把你的主机名写进去，也可以不写，取决于你的安全需求。这个例子中，我们写了。

我们现在自己签署这个请求：

```
$ openssl x509 -req -in request.pem -signkey laptop.private -out \
  laptop.public
Signature ok
subject=/C=NL/L=Delft/O=Linux Advanced Routing & Traffic \
  Control/OU=laptop/CN=bert hubert/Email=ahu@ds9a.nl
Getting Private key
```

现在，" request.pem " 这个文件已经没用了，可以删除。

在你需要证书的每台机器上都重复上述过程。你现在就可以放心地发布你的 " *.public " 文件了，但是一定要保证 " *.private " 是保密的！

7.2.3.2. 设置并启动

我们一旦拥有了一对公钥和私钥，就可以告诉 racoon 去使用它们了。

现在我们回到上面的配置中的两台机器，10.0.0.11 (upstairs)和 10.0.0.216(laptop)。

在 10.0.0.11 上的 racoon.conf 中，我们添加：

```
path certificate "/usr/local/etc/racoon/certs";

remote 10.0.0.216
{
exchange_mode aggressive,main;
my_identifier asn1dn;
peers_identifier asn1dn;

certificate_type x509 "upstairs.public" "upstairs.private";

peers_certfile "laptop.public";
proposal {
  encryption_algorithm 3des;
  hash_algorithm sha1;
  authentication_method rsasig;
  dh_group 2 ;
  }
}
```

它们告诉 racoon：证书可以在/usr/local/etc/racoon/certs/那里找到。而且还包含了专门为 10.0.0.216 而写的配置项。

包含“ asn1dn ”的行告诉 racoon，本端和对端的标识都从公钥中提取。也就是上面输出的“ `subject=/C=NL/L=Delft/O=Linux Advanced Routing & Traffic Control/OU=laptop/CN=bert hubert/Email=ahu@ds9a.nl` ”。

“ **certificate_type** ”那一行配置了本地的公钥和私钥。“ **peers_certfile** ”这行告诉 racoon 读取名叫“ laptop.public ”的文件取得对端的公钥。

“ **proposal** ”这一段与你以前看到的基本一致，除了“ **authentication_method** ”的值变成了“ **rsasig** ”，意思是使用 RSA 公钥/私钥对。

在 10.0.0.216 上面的配置文件与上面的是完全镜像关系，没有其它改变：

```
path certificate "/usr/local/etc/racoon/certs";

remote 10.0.0.11
{
  exchange_mode aggressive,main;
  my_identifier asn1dn;
  peers_identifier asn1dn;
  certificate_type x509 "laptop.public" "laptop.private";
  peers_certfile "upstairs.public";

  proposal {
    encryption_algorithm 3des;
    hash_algorithm sha1;
    authentication_method rsasig;
    dh_group 2 ;
  }
}
```

现在，我们已经把两台机器的配置文件改好了，然后就应该把证书文件拷贝到正确的位置。“ upstairs ”这台机器需要往/usr/local/etc/racoon/certs 中放入 upstairs.private、upstairs.public 和 laptop.public。请确认这个目录属于 root，且属性为 0600，否则 racoon 会拒绝使用！

“ laptop ”这台机器需要往/usr/local/etc/racoon/certs 中放入 laptop.private、laptop.public 和 upstairs.public。也就是说，每台机器都需要本端的公钥和私钥，

以及对端的公钥。

确认一下已经写好了 SP(执行在 7.2.2 中提到的 spdadd)。然后启动 racoon，就应该可以工作了。

7.2.3.3. 如何安全地建立隧道

为了与对端建立安全的通讯，我们必须交换公钥。公钥没必要保密，重要的是要保证它不被替换。换句话说，要确保没有“中间人”。

为了简化这个工作，OpenSSL 提供了“digest”命令：

```
$ openssl dgst upstairs.public
MD5(upstairs.public)= 78a3bddafb4d681c1ca8ed4d23da4ff1
```

现在我们要做的就是检验一下对方是否能够得到相同的 MD5 散列值。这可以通过真实地接触来完成，也可以通过电话，但是一定不要与公钥放在同一封电子邮件里发送！

另一个办法是通过一个可信的第三方（CA）来实现。这个 CA 会为你的密钥进行签名，而不是象上面那样由我们自己签名。

7.3. IPSEC 隧道

迄今为止，我们只是认识了 IPSEC 的“transport”(透明)模式，也就是通讯的两端都能够直接理解 IPSEC。这不能代表所有的情况，有时候我们只需要路由器理解 IPSEC，路由器后面的机器利用它们进行通讯。这就是所谓的“tunnel mode”（隧道模式）。

设置这个极其简单。如果想通过 10.0.0.216 与 10.0.0.11 建立的隧道来传输从 10.0.0.0/24 到 130.161.0.0/16 的数据包，按下面配置就可以：

```
#!/sbin/setkey -f
flush;
spdflush;

add 10.0.0.216 10.0.0.11 esp 34501
-m tunnel
-E 3des-cbc "123456789012123456789012";

spdadd 10.0.0.0/24 130.161.0.0/16 any -P out ipsec
           esp/tunnel/10.0.0.216-10.0.0.11/require;
```

注意。“-m tunnel”是关键！这里首先配置了一个隧道两端(10.0.0.216 与 10.0.0.11)使用 ESP 的 SA。

然后配置了实际的隧道。它指示内核，对于从 10.0.0.0/24 到 130.161.0.0/16 的数据包需要加密。而且这些数据包被发往 10.0.0.11。

10.0.0.11 也需要相同的配置：

```
#!/sbin/setkey -f
flush;
```

```
spdflush;

add 10.0.0.216 10.0.0.11 esp 34501
-m tunnel
-E 3des-cbc "123456789012123456789012";

spdadd 10.0.0.0/24 130.161.0.0/16 any -P in ipsec
           esp/tunnel/10.0.0.216-10.0.0.11/require;
```

注意，与上面基本一样，除了把“ -P out ”换成了“ -P in ”。就象先前的例子一样，我们只配置了单向的传输。完整地实现双向传输就留给读者自己研究实现吧。

这种配置的另一个更直观的名字叫做“ ESP 代理(proxy ESP) ”。

- IPSEC 隧道需要内核能够进行 IP 转发！

7.4. 其它 IPSEC 软件

Thomas Walpuski 报告说它已经写了一个补丁，可以让 OpenBSD 的 isakpmd 与 Linux 2.5 的 IPSEC 协同工作。可以在这个网页找到。他指出 isakpmd 在 Linux 上仅仅需要 `libkeynote` 支持。根据他的说法，在 Linux 2.5.59 上面工作得很好。

isakpmd 与前面提到的 racoon 由很大的不同，但是很多人喜欢用它，可以在这里找到。这里有 OpenBSD CVS。Thomas 还为那些不习惯用 CVS 或者 patch 的人们制作了一个 tarball。

7.5. IPSEC 与其它系统的互操作

求助: Write this

7.5.1. Windows

第 8 章 多播路由

求助: Editor Vacancy!

Multicast-HOWTO 已经很古老了(相对而言) ，而且不够准确甚至会因此而误导读者。

在你开始进行多播路由之前，你需要重新配置你的 Linux 内核来支持你想实现的多播路由类型。这一步需要你来决定使用何种类型的多播路由。基本上有这么四种：DVMRP (RIP 单播协议的多播版本)，MOSPF(同理，只不过是 OSPF)，PIM-SM ("Protocol Independent Multicasting - Sparse Mode"，协议无关多播-稀疏模式，它假定任意多播组的用户都是 spread out 的，而不是 clumped 的)和 PIM-DM (同理，只不过是“ 密集模式 ”，它假定同一个多播组的用户适当地 clumps 在一起).

在 Linux 内核中，你会注意到并没有这些选项。这是因为这些协议本身是由路由程序负责处理的，比如 Zebra、mrouted 或者 pimd。然而你仍然应该对于你要使用那种方案有一个明确的主意，以便选择正确的内核选项。

无论哪种多播路由，你一定都要启用“ multicasting ”和“ multicast routing ”选项。对于 DVMRP 和 MOSPF 这就够了。如果你想使用 PIM，你必须还要启用 PIMv1 或者 PIMv2 选项，具体用哪个取决于你的网络究竟使用 PIM 协议的哪一个版本。

当你把那些都想清楚、编译了新内核之后，重启的时候应该能够看到 IP 协议的列表中包括了 IGMP。这是用来管理多播组的协议。虽然第 3 版业已存在并归档，但截止到写这篇文档时为止，Linux 只支持 IGMP 的第 1 版和第 2 版。但这并不会太多地影响我们，因为 IGMPv3 还太新，并没有看到多少能够用到 v3 特有功能的应用。因为，用 IGMP 处理组，仅会使用到连最简单的 IGMP 版本中都会包含的基本功能。绝大部分应该是 IGMPv2，虽然仍能接触到 IGMPv1。

到此为止，一切都好。我们已经启用了多播。现在，我们得告诉内核，做点实在的事情了——启动路由。也就是说象路由表中添加多播子网：

```
ip route add 224.0.0.0/4 dev eth0
```

(当然，我们假定你要通过 eth0 进行多播。你要根据你的情况选择设备。)

现在，启动 Linux 的包转发...

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

在这里，你可能想知道是否起了作用。所以我们 ping 一下缺省组 224.0.0.1，看看有没有人在。在你的 LAN 上所有配置并启用了多播的机器都*应该*予以回应，其他机器则不会。但你会注意到，没有任何一台机器回应的时候声明自己是 224.0.0.1，多么令人惊奇☺！因为这是一个组地址（对于接收者来说是“ 广播 ”)，所以组中的所有成员都用它们的地址来回应，而不是用组地址来回应。

```
ping -c 2 224.0.0.1
```

到此，你已经可以实现真正的多播路由了。好的，假定你需要在两个网络间进行路由。

(To Be Continued!)

第 9 章 带宽管理的队列规定

现在，当我搞清楚这一切之后，我真的大吃一惊。Linux 2.2/2.4 完全能象那些最高端的专用带宽管理系统一样来管理带宽。甚至比帧中继和 ATM 还要优秀。

为避免概念混乱，**tc** 采用如下规定来描述带宽：

```
mbps = 1024 kbps = 1024 * 1024 bps => byte/s
mbit = 1024 kbit => kilo bit/s.
mb = 1024 kb = 1024 * 1024 b => byte
mbit = 1024 kbit => kilo bit.
```

内定：数字以 bps 和 b 方式储存。

但当 **tc** 输出速率时，使用如下表示：

```
1Mbit = 1024 Kbit = 1024 * 1024 bps => byte/s
```

9.1. 解释队列和队列规定

利用队列，我们决定了数据被**发送**的方式。必须认识到，我们只能对发送数据进行整形。

根据 Internet 的工作方式，我们无法直接控制别人向我们发送什么数据。有点象我们家里的信报箱，你不可能控制全世界，联系每一个人，修改别人对你发送邮件的数量。

然而，Internet 主要依靠 TCP/IP，它的一些特性很有用。因为 TCP/IP 没办法知道两个主机之间的网络容量，所以它会试图越来越快地发送数据(所谓的“慢起技术”)，当因为网络容量不够而开始丢失数据时，再放慢速度。实际情况要比这种方法更聪明，我们以后再讨论。

这就象当你尚未读完一半邮件时，希望别人停止给你寄信。与现实世界不同，在 Internet 上可以做到这一点。（译注：这个例子并不恰当，TCP/IP 的这种机制并不是在网络层实现的，而是要靠传输层的 TCP 协议）

如果你有一个路由器，并且希望能够防止某些主机下载速度太快，你需要在你路由器的*内网卡*——也就是向你的网内主机发送数据包的网卡——上进行流量整形。

你还要保证你正在控制的是瓶颈环节。如果你有一个 100M 以太网卡，而你的路由器的链路速度是 256k，你必须保证你发送的数据量没有超过路由器的处理能力。否则，就是路由器在控制链路和对带宽进行整形，而不是你。可以说，我们需要拥有的队列必须是一系列链路中最慢的环节。幸运的是这很容易。

9.2. 简单的无类队列规定

如前所述，利用队列，我们决定了数据被**发送**的方式。无类队列规定就是那样，能够接受数据和重新编排、延迟或丢弃数据包。

这可以用作对于整个网卡的流量进行整形，而不细分各种情况。在我们进一步学习分类的队列规定之前，理解这部分是必不可少的！

最广泛应用的规定是 pfifo_fast 队列规定，因为它是缺省配置。这也解释了为什么其它那些复杂的功能为何如此健壮，因为那些都与缺省配置相似，只不过是其他类型的队列而已。

每种队列都有它们各自的优势和弱点。

9.2.1. pfifo_fast

这个队列的特点就象它的名字——先进先出（FIFO），也就是说没有任何数据包被特殊对待。至少不是非常特殊。这个队列有 3 个所谓的“频道”。FIFO 规则应用于每一个频道。并且：如果在 0 频道有数据包等待发送，1 频道的包就不会被处理，1 频道和 2 频道之间的关系也是如此。

内核遵照数据包的 TOS 标记，把带有“最小延迟”标记的包放进 0 频道。

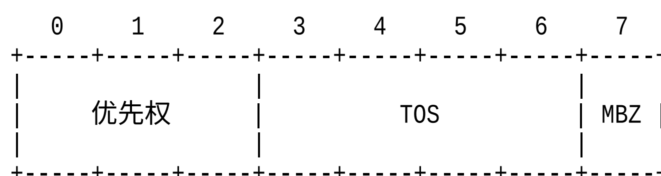
不要把这个无类的简单队列规定与分类的 PRIO 相混淆！虽然它们的行为有些类似，但对于无类的 pfifo_fast 而言，你不能使用 tc 命令向其中添加其它的队列规定。

9.2.1.1. 参数与使用

pfifo_fast 队列规定作为硬性的缺省设置，你不能对它进行配置。它缺省是这样配置的：

priomap：

内核规定，根据数据包的优先权情况，对应相应的频道。这个对应是根据数据包的 TOS 字节进行的。TOS 看上去是这样的：



```
   0     1     2     3     4     5     6     7
+-----+-----+-----+-----+-----+-----+-----+-----+
|                 |                       |     |
|     优先权      |          TOS          | MBZ |
|                 |                       |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

TOS 字段的 4 个 bit 是如下定义的：

```
二进制    十进制   意义
-----------------------------------------
1000      8        最小延迟 (md)
0100      4        最大 throughput (mt)
0010      2        最大可靠性 (mr)
```

```
0001     1        最小成本 (mmc)
0000     0        正常服务
```

因为在这 4bit 的后面还有一个 bit，所以 TOS 字段的实际值是上述值的 2 倍。(Tcpdump -v -v 可以让你看到整个 TOS 字段的情况，而不仅仅是这 4 个 bit）也就是你在下表的第一列看到的值：

```
TOS     Bits  意义                    Linux 优先权     频道
------------------------------------------------------------
0x0     0     正常服务                0 最好效果       1
0x2     1     最小成本(mmc)           1 填充           2
0x4     2     最大可靠性(mr)          0 最好效果       1
0x6     3     mmc+mr                  0 最好效果       1
0x8     4     最大吞吐量(mt)          2 大量传输       2
0xa     5     mmc+mt                  2 大量传输       2
0xc     6     mr+mt                   2 大量传输       2
0xe     7     mmc+mr+mt               2 大量传输       2
0x10    8     最小延迟(md)            6 交互           0
0x12    9     mmc+md                  6 交互           0
0x14    10    mr+md                   6 交互           0
0x16    11    mmc+mr+md               6 交互           0
0x18    12    mt+md                   4 交互+大量传输  1
0x1a    13    mmc+mt+md               4 交互+大量传输  1
0x1c    14    mr+mt+md                4 交互+大量传输  1
0x1e    15    mmc+mr+mt+md            4 交互+大量传输  1
```

很多的数字。第二列写着与 4 个 TOS 位相关的数值，接着是它们的意义。比如，15 表示一个数据包要求最小成本、最大可靠性、最大吞吐量和最小延迟。我想称之为“人代会车队”。[译者按：原作为‘荷兰数据包’]

第四列写出了 Linux 内核对于 TOS 位的理解，并表明了它们对应哪种优先权。

最后一列表明缺省的权限图。在命令行里，缺省的权限图应该是：

```
1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1
```

也就是说，比如优先权 4 将被映射到 1 频道。权限图允许你列出更高的优先权值(>7)，它们不对应 TOS 映射，但是有其它的意图。

下表来自 RFC 1349，告诉你应用程序可能如何设置它们的 TOS：

TELNET		1000	(minimize delay)
FTP	控制	1000	(minimize delay)
	数据	0100	(maximize throughput)
TFTP		1000	(minimize delay)
SMTP	命令阶段	1000	(minimize delay)
	数据阶段	0100	(maximize throughput)
Domain Name Service	UDP 查询	1000	(minimize delay)
	TCP 查询	0000	
	区域传输	0100	(maximize throughput)
NNTP		0001	(minimize monetary cost)
ICMP	报错	0000	
	请求	0000	(mostly)
	响应	<与请求相同>	(mostly)

txqueuelen

队列的长度来自网卡的配置，你可以用 ifconfig 和 ip 命令修改。如设置队列长度为 10，执行：ifconfig eth0 txqueuelen 10

你不能用 tc 命令设置这个！

9.2.2. 令牌桶过滤器(TBF)

令牌桶过滤器(TBF)是一个简单的队列规定：只允许以不超过事先设定的速率到来的数据包通过，但可能允许短暂突发流量朝过设定值。

TBF 很精确,对于网络和处理器的影响都很小。所以如果您想对一个网卡限速，它应该成为您的第一选择！

TBF 的实现在于一个缓冲器(桶)，不断地被一些叫做“令牌”的虚拟数据以特定速率填充着。 (token rate)。桶最重要的参数就是它的大小，也就是它能够存储令牌的数量。

每个到来的令牌从数据队列中收集一个数据包，然后从桶中被删除。这个算法关联到两个流上——令牌流和数据流，于是我们得到 3 种情景：

- 数据流以*等于*令牌流的速率到达 TBF。这种情况下，每个到来的数据包都能对应一个令牌，然后无延迟地通过队列。
- 数据流以*小于*令牌流的速度到达 TBF。通过队列的数据包只消耗了一部分令牌，剩下的令牌会在桶里积累下来，直到桶被装满。剩下的令牌可以在需要以高于令牌流速率发送数据流的时候消耗掉，这种情况下会发生突发传输。
- 数据流以*大于*令牌流的速率到达 TBF。这意味着桶里的令牌很快就会被耗尽。导致 TBF 中断一段时间，称为“越限”。如果数据包持续到来，将发生丢包。

最后一种情景非常重要，因为它可以用来对数据通过过滤器的速率进行整形。

令牌的积累可以导致越限的数据进行短时间的突发传输而不必丢包，但是持续越限的话会导致传输延迟直至丢包。

请注意，实际的实现是针对数据的字节数进行的，而不是针对数据包进行的。

9.2.2.1. 参数与使用

即使如此，你还是可能需要进行修改，TBF 提供了一些可调控的参数。第一个参数永远可用：

limit/latency

limit 确定最多有多少数据（字节数）在队列中等待可用令牌。你也可以通过设置 latency 参数来指定这个参数，latency 参数确定了一个包在 TBF 中等待传输的最长等待时间。后者计算决定桶的大小、速率和峰值速率。

burst/buffer/maxburst

桶的大小，以字节计。这个参数指定了最多可以有多少个令牌能够即刻被使用。通常，管理的带宽越大，需要的缓冲器就越大。在 Intel 体系上，10 兆 bit/s 的速率需要至少 10k 字节的缓冲区才能达到期望的速率。

如果你的缓冲区太小，就会导致到达的令牌没有地方放（桶满了），这会导致潜在的丢包。

mpu

一个零长度的包并不是不耗费带宽。比如以太网，数据帧不会小于 64 字节。Mpu(Minimum Packet Unit，最小分组单位)决定了令牌的最低消耗。

rate

速度操纵杆。参见上面的 limits！

如果桶里存在令牌而且允许没有令牌，相当于不限制速率(缺省情况)。If the bucket contains tokens and is allowed to empty, by default it does so at infinite speed. 如果不希望这样，可以调整入下参数：

peakrate

如果有可用的令牌，数据包一旦到来就会立刻被发送出去，就象光速一样。那可能并不是你希望的，特别是你有一个比较大的桶的时候。

峰值速率可以用来指定令牌以多块的速度被删除。用书面语言来说，就是：释放一个数据包，但后等待足够的时间后再释放下一个。我们通过计算等待时间来控制峰值速率

然而，由于 UNIX 定时器的分辨率是 10 毫秒，如果平均包长 10k bit，我们的峰值速率被限制在了 1Mbps。

mtu/minburst

但是如果你的常规速率比较高，1Mbps 的峰值速率对我们就没有什么价值。要实现更高的峰值速率，可以在一个时钟周期内发送多个数据包。最有效的办法就是：再创建一个令牌桶！

这第二个令牌桶缺省情况下为一个单个的数据包，并非一个真正的桶。

要计算峰值速率，用 mtu 乘以 100 就行了。 (应该说是乘以 HZ 数，Intel 体系上是 100，Alpha 体系上是 1024)

9.2.2.2. 配置范例

这是一个非常简单而实用的例子：

```
# tc qdisc add dev ppp0 root tbf rate 220kbit latency 50ms burst 1540
```

为什么它很实用呢？如果你有一个队列较长的网络设备，比如 DSL modem 或者 cable modem 什么的，并通过一个快速设备(如以太网卡)与之相连，你会发现上

载数据绝对会破坏交互性。

这是因为上载数据会充满 modem 的队列，而这个队列为了改善上载数据的吞吐量而设置的特别大。但这并不是你需要的，你可能为了提高交互性而需要一个不太大的队列。也就是说你希望在发送数据的时候干点别的事情。

上面的一行命令并非直接影响了 modem 中的队列，而是通过控制 Linux 中的队列而放慢了发送数据的速度。

把 220kbit 修改为你实际的上载速度再减去几个百分点。如果你的 modem 确实很快，就把“ burst ”值提高一点。

9.2.3. 随机公平队列(SFQ)

SFQ(Stochastic Fairness Queueing，随机公平队列)是公平队列算法家族中的一个简单实现。它的精确性不如其它的方法，但是它在实现高度公平的同时，需要的计算量却很少。

SFQ 的关键词是“ 会话 ”(或称作“ 流 ”) ，主要针对一个 TCP 会话或者 UDP 流。流量被分成相当多数量的 FIFO 队列中，每个队列对应一个会话。数据按照简单轮转的方式发送, 每个会话都按顺序得到发送机会。

这种方式非常公平，保证了每一个会话都不会没其它会话所淹没。SFQ 之所以被称为“ 随机 ”，是因为它并不是真的为每一个会话创建一个队列，而是使用一个散列算法，把所有的会话映射到有限的几个队列中去。

因为使用了散列，所以可能多个会话分配在同一个队列里，从而需要共享发包的机会，也就是共享带宽。为了不让这种效应太明显，SFQ 会频繁地改变散列算法，以便把这种效应控制在几秒钟之内。

有很重要的一点需要声明：只有当你的出口网卡确实已经挤满了的时候，SFQ 才会起作用！否则在你的 Linux 机器中根本就不会有队列，SFQ 也就不会起作用。稍后我们会描述如何把 SFQ 与其它的队列规定结合在一起，以保证两种情况下都比较好的结果。

特别地，在你使用 DSL modem 或者 cable modem 的以太网卡上设置 SFQ 而不进行任何进一步地流量整形是无谋的！

9.2.3.1. 参数与使用

SFQ 基本上不需要手工调整：

perturb

    多少秒后重新配置一次散列算法。如果取消设置，散列算法将永远不会重新配置（不建议这样做)。10 秒应该是一个合适的值。

quantum

一个流至少要传输多少字节后才切换到下一个队列。却省设置为一个最大包的长度(MTU 的大小)。不要设置这个数值低于 MTU！

9.2.3.2. 配置范例

如果你有一个网卡，它的链路速度与实际可用速率一致——比如一个电话 MODEM——如下配置可以提高公平性：

```
# tc qdisc add dev ppp0 root sfq perturb 10
# tc -s -d qdisc ls
qdisc sfq 800c: dev ppp0 quantum 1514b limit 128p flows 128/1024 perturb 10sec
 Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

“ 800c: ” 这个号码是系统自动分配的一个句柄号，“ limit ” 意思是这个队列中可以有 128 个数据包排队等待。一共可以有 1024 个散列目标可以用于速率审计，而其中 128 个可以同时激活。(no more packets fit in the queue!)每隔 10 秒种散列算法更换一次。

9.3. 关于什么时候用哪种队列的建议

总之，我们有几种简单的队列，分别使用排序、限速和丢包等手段来进行流量整形。

下列提示可以帮你决定使用哪一种队列。涉及到了*第 14 章* 所描述的的一些队列规定：

- 如果想单纯地降低出口速率，使用令牌桶过滤器。调整桶的配置后可用于控制很高的带宽。
- 如果你的链路已经塞满了，而你想保证不会有某一个会话独占出口带宽，使用随机公平队列。
- 如果你有一个很大的骨干带宽，并且了解了相关技术后，可以考虑前向随机丢包(参见 “ 高级 ” 那一章)。
- 如果希望对入口流量进行“ 整形 ”(不是转发流量)，可使用入口流量策略，注意，这不是真正的 “ 整形 ”。
- 如果你正在转发数据包，在数据流出的网卡上应用 TBF。除非你希望让数据包从多个网卡流出，也就是说入口网卡起决定性作用的时候，还是使用入口策略。
- 如果你并不希望进行流量整形，只是想看看你的网卡是否有比较高的负载而需要使用队列，使用 pfifo 队列（不是 pfifo_fast）。它缺乏内部频道但是可以统计 backlog。
- 最后，你可以进行所谓的 “ 社交整形 ”。你不能通过技术手段解决一切问题。用户的经验技巧永远是不友善的。正确而友好的措辞可能帮助你的正确地分配带宽！

9.4. 术语

为了正确地理解更多的复杂配置，有必要先解释一些概念。由于这个主题的历史不长和其本身的复杂性，人们经常在说同一件事的时候使用各种词汇。

以下来自 draft-ietf-diffserv-model-06.txt，*Diffserv 路由器的建议管理模型。* 可以在以下地址找到：

http://www.ietf.org/internet-drafts/draft-ietf-diffserv-model-06.txt.

关于这些词语的严格定义请参考这个文档。

队列规定

> 管理设备输入(ingress)或输出(egress)的一个算法。

无类的队列规定

> 一个内部不包含可配置子类的队列规定。

分类的队列规定

> 一个分类的队列规定内可一包含更多的类。其中每个类又进一步地包含一个队列规定，这个队列规定可以是分类的，也可以是无类的。根据这个定义，严格地说 pfifo_fast 算是分类的，因为它实际上包含 3 个频道(实际上可以认为是子类)。然而从用户的角度来看它是无类的，因为其内部的子类无法用 tc 工具进行配置。

类

> 一个分类的队列规定可以拥有很多类，类内包含队列规定。

分类器

> 每个分类的队列规定都需要决定什么样的包使用什么类进行发送。分类器就是做这个用的。

过滤器

> 分类是通过过滤器完成的。一个过滤器包含若干的匹配条件，如果符合匹配条件，就按此过滤器分类。

调度

> 在分类器的帮助下，一个队列规定可以裁定某些数据包可以排在其他数据包之前发送。这种处理叫做“ 调度 ”，比如此前提到的 pfifo_fast 就是这样的。调度也可以叫做“ 重排序 ”，但这样容易混乱。

整形

> 在一个数据包发送之前进行适当的延迟，以免超过事先规定好的最大速率，这种处理叫做“ 整形 ”。整形在 egress 处进行。习惯上，通过丢包来降速也经常被称为整形。

策略

通过延迟或是丢弃数据包来保证流量不超过事先规定的带宽。在 Linux, 里，策略总是规定丢弃数据包而不是延迟。即，不存在 ingress 队列。

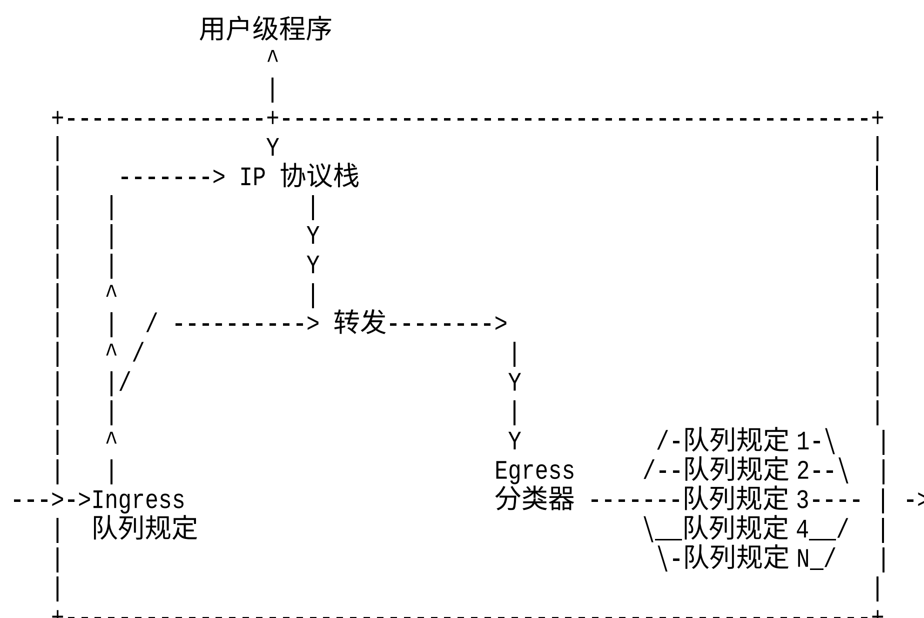
Work-Conserving

对于一个 work-conserving 队列规定，如果得到一个数据包，它总是立刻对它进行分发。换句话说，只要网卡(egress 队列规定)允许，它就不会延迟数据包的发送。

non-Work-Conserving

有些队列——比如令牌桶过滤器——可能需要暂时停止发包以实现限制带宽。也就是说它们有时候即使有数据包需要处理，也可能拒绝发送。

现在我们简单了解了一些术语，让我们看看他们的位置：



感谢 Jamal Hadi Salim 制作的 ASCII 字符图像。

整个大方框表示内核。最左面的箭头表示从网络上进入机器的数据包。它们进入 Ingress 队列规定，并有可能被某些过滤器丢弃。即所谓策略。

这些是很早就发生的（在进入内核更深的部分之前）。这样早地丢弃数据有利于节省 CPU 时间。

数据包顺利通过的话，如果它是发往本地进程的，就会进入 IP 协议栈处理并提交给该进程。如果它需要转发而不是进入本地进程，就会发往 egress。本地进程也可以发送数据，交给 Egress 分类器。

然后经过审查，并放入若干队列规定中的一个进行排队。这个过程叫做“ 入队 ”。在不进行任何配置的情况下，只有一个 egress 队列规定——pfifo_fast——总是接收数据包。

数据包进入队列后，就等待内核处理并通过某网卡发送。这个过程叫做“ 出队 ”。

这张图仅仅表示了机器上只有一块网卡的情况，图中的箭头不能代表所有情况。每块网卡都有它自己的 ingress 和 egress。

9.5. 分类的队列规定

如果你有多种数据流需要进行区别对待，分类的队列规定就非常有用了。众多分类的队列规定中的一种——CBQ(Class Based Queueing，基于类的队列)——经常被提起，以至于造成大家认为 CBQ 就是鉴别队列是否分类的标准，这是不对的。

CBQ 不过是家族中最大的孩子而已，同时也是最复杂的。它并不能为你做所有的事情。对于某些人而言这有些不可思议，因为他们受" sendmail 效应 "影响较深，总是认为只要是复杂的并且没有文档的技术肯定是最好的。

9.5.1. 分类的队列规定及其类中的数据流向

一旦数据包进入一个分类的队列规定，它就得被送到某一个类中——也就是需要分类。对数据包进行分类的工具是过滤器。一定要记住：" 分类器 "是从队列规定内部调用的，而不是从别处。

过滤器会返回一个决定，队列规定就根据这个决定把数据包送入相应的类进行排队。每个子类都可以再次使用它们的过滤器进行进一步的分类。直到不需要进一步分类时，数据包才进入该类包含的队列规定排队。

除了能够包含其它队列规定之外，绝大多数分类的队列规定黑能够流量整形。这对于需要同时进行调度(如使用 SFQ)和流量控制的场合非常有用。如果你用一个高速网卡(比如以太网卡)连接一个低速设备(比如 cable modem 或者 ADSL modem)时，也可以应用。

如果你仅仅使用 SFQ，那什么用也没有。因为数据包进、出路由器时没有任何延迟。虽然你的输出网卡远远快于实际连接速率，但路由器中却没有队列可以调度。

9.5.2. 队列规定家族：根、句柄、兄弟和父辈

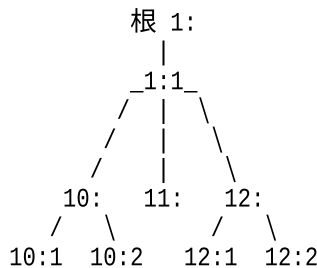
每块网卡都有一个出口" 根队列规定 "，缺省情况下是前面提到的 pfifo_fast 队列规定。每个队列规定都指定一个句柄，以便以后的配置语句能够引用这个队列规定。除了出口队列规定之外，每块网卡还有一个入口,以便 policies 进入的数据流。

队列规定的句柄有两个部分：一个主号码和一个次号码。习惯上把根队列规定称为" 1: "，等价于" 1:0 "。队列规定的次号码永远是 0。

类的主号码必须与它们父辈的主号码一致。

9.5.2.1. 如何用过滤器进行分类

下图给出一个典型的分层关系：



不要误解这张图！你千万**不要**想象内核处在树的顶点而下面部分是网络。数据包是在根队列规定处入队和出队的，而内核只同根打交道。

一个数据包可能是按照下面这个链状流程进行分类的：

1: -> 1:1 -> 12: -> 12:2

数据包现在应该处于 12:2 下属的某个队列规定中的某个队列中。在这个例子中，树的每个节点都附带着一个过滤器，用来选择下一步进入哪个分支。这样比较直观。然而，这样也是允许的：

1: -> 12:2

也就是说，根所附带的一个过滤器要求把数据包直接交给 12:2。

9.5.2.2. 数据包如何出队并交给硬件

当内核决定把一个数据包发给网卡的时候，根队列规定 1:会得到一个出队请求，然后把它传给 1:1，然后依次传给 10:、11:和 12:，which each query their siblings，然后试图从它们中进行 dequeue()操作。也就是说，内和需要遍历整颗树，因为只有 12:2 中才有这个数据包。

换句话说，类及其兄弟**仅仅**与其“ 父队列规定 ”进行交谈，而不会与网卡进行交谈。只有根队列规定才能由内核进行出队操作！

更进一步，任何类的出队操作都不会比它们的父类更快。这恰恰是你所需要的：我们可以把 SFQ 作为一个子类，放到一个可以进行流量整形的父类中，从而能够同时得到 SFQ 的调度功能和其父类的流量整形功能。

9.5.3. PRIO 队列规定

PRIO 队列规定并不进行整形，它仅仅根据你配置的过滤器把流量进一步细分。你可以认为 PRIO 队列规定是 pfifo_fast 的一种衍生物，区别在每个频道都是一个单独的类，而非简单的 FIFO。

当数据包进入 PRIO 队列规定后，将根据你给定的过滤器设置选择一个类。缺省情况下有三个类，这些类仅包含纯 FIFO 队列规定而没有更多的内部结构。你可以把它们替换成你需要的任何队列规定。

每当有一个数据包需要出队时，首先处理:1 类。只有当标号更小的类中没有需要处理的包时，才会标号大的类。

当你希望不仅仅依靠包的 TOS，而是想使用 tc 所提供的更强大的功能来进行数据包的优先权划分时，可以使用这个队列规定。它也可以包含更多的队列规定，而 pfifo_fast 却只能包含简单的 fifo 队列规定。

因为它不进行整形，所以使用时与 SFQ 有相同的考虑：要么确保这个网卡的带宽确实已经占满，要么把它包含在一个能够整形的分类的队列规定的内部。后者几乎涵盖了所有 cable modems 和 DSL 设备。

严格地说，PRIO 队列规定是一种 Work-Conserving 调度。

9.5.3.1. PRIO 的参数与使用

tc 识别下列参数：

bands

创建频道的数目。每个频道实际上就是一个类。如果你修改了这个数值，你必须同时修改：

priomap

如果你不给 tc 提供任何过滤器，PRIO 队列规定将参考 TC_PRIO 的优先级来决定如何给数据包入队。

它的行为就像前面提到过的 pfifo_fast 队列规定，关于细节参考前面章节。

频道是类，缺省情况下命名为主标号:1 到主标号:3。如果你的 PRIO 队列规定是 12:，把数据包过滤到 12:1 将得到最高优先级。

注意：0 频道的次标号是 1！1 频道的次标号是 2，以此类推。

9.5.3.2. 配置范例

我们想创建这个树：

```
      root 1: prio
        /   |   \
      1:1  1:2  1:3
       |    |    |
      10:  20:  30:
      sfq  tbf  sfq
 band  0    1    2
```

大批量数据使用 30:，交互数据使用 20:或 10:。

命令如下：

```
# tc qdisc add dev eth0 root handle 1: prio
## 这个命令立即创建了类： 1:1, 1:2, 1:3

# tc qdisc add dev eth0 parent 1:1 handle 10: sfq
# tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate 20kbit buffer 1600 limit 3000
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
```

现在，我们看看结果如何：

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 132 bytes 2 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 174 bytes 3 pkts (dropped 0, overlimits 0)
```

如你所见，0 频道已经有了一些流量，运行这个命令之后发送了一个包！

现在我们来点大批量数据传输（使用能够正确设置 TOS 标记的工具）：

```
# scp tc ahu@10.0.0.11:./
ahu@10.0.0.11's password:
tc                   100% |*****************************|   353 KB    00:00
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 2230 bytes 31 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 389140 bytes 326 pkts (dropped 0, overlimits 0)
```

如你所见，所有的流量都是经过 30:处理的，优先权最低。现在我们验证一下交互数据传输经过更高优先级的频道，我们生成一些交互数据传输：

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 14926 bytes 193 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 401836 bytes 488 pkts (dropped 0, overlimits 0)
```

正常——所有额外的流量都是经 10:这个更高优先级的队列规定处理的。与先前的整个 scp 不同，没有数据经过最低优先级的队列规定。

9.5.4. 著名的 CBQ 队列规定

如前所述，CBQ 是最复杂、最琐碎、最难以理解、最刁钻的队列规定。这并不是因为其作者的恶毒或者不称职，而是因为 CBQ 算法本身的不精确，而且与 Linux 的内在机制不协调造成的。

除了可以分类之外，CBQ 也是一个整形器，但是从表面上看来工作得并不好。它应该是这样的：如果你试图把一个 10Mbps 的连接整形成 1Mbps 的速率，就应该让链路 90%的时间处于闲置状态，必要的话我们就强制，以保证 90%的闲置时间。

但闲置时间的测量非常困难，所以 CBQ 就采用了它一个近似值——来自硬件层的两个传输请求之间的毫秒数——来代替它。这个参数可以近似地表征这个链路的繁忙程度。

这样做相当慎重，而且不一定能够得到正确的结论。比如，由于驱动程序方面或者其它原因造成一块网卡的实际传输速率不能够达到它的标称速率，该怎么办？由于总线设计的原因，PCMCIA 网卡永远也不会达到 100Mbps。那么我们该怎么计算闲置时间呢？

如果我们引入非物理网卡——像 PPPoE、PPTP——情况会变得更糟糕。因为相当一部分有效带宽耗费在了链路维护上。

那些实现了测量的人们都发现 CBQ 总不是非常精确甚至完全失去了其本来意义。

但是，在很多场合下它还是能够很好地工作。根据下面的文档，你应该能够较好地配置 CBQ 来解决答多数问题。

9.5.4.1. CBQ 整形的细节

如前所述，CBQ 的工作机制是确认链路的闲置时间足够长，以达到降低链路实际带宽的目的。为此，它要计算两个数据包的平均发送间隔。

操作期间，有效闲置时间的测量使用 EWMA(exponential weighted moving average, 指数加权移动均值)算法，也就是说最近处理的数据包的权值比以前的数据包按指数增加。UNIX 的平均负载也是这样算出来的。

计算出来的平均时间值减去 EWMA 测量值，得出的结果叫做“avgidle”。最佳的链路负载情况下，这个值应当是 0：数据包严格按照计算出来的时间间隔到来。

在一个过载的链路上，avgidle 值应当是负的。如果这个负值太严重，CBQ 就会暂时禁止发包，称为“overlimit”(越限)。

相反地，一个闲置的链路应该有很大的 avgidle 值,这样闲置几个小时后，会造成链路允许非常大的带宽通过。为了避免这种局面，我们用 maxidle 来限制 avgidle 的值不能太大。

理论上讲，如果发生越限，CBQ 就会禁止发包一段时间(长度就是事先计算出来的传输数据包之间的时间间隔)，然后通过一个数据包后再次禁止发包。但是最好参照一下下面的 minburst 参数。

下面是配置整形时需要指定的一些参数：

avpkt

　　平均包大小，以字节计。计算 maxidle 时需要,maxidle 从 maxburst 得出。

bandwidth

网卡的物理带宽，用来计算闲置时间。

cell

一个数据包被发送出去的时间可以是基于包长度而阶梯增长的。一个 800 字节的包和一个 806 字节的包可以认为耗费相同的时间。也就是说它设置时间粒度。通常设置为 8，必须是 2 的整数次幂。

maxburst

这个参数的值决定了计算 maxidle 所使用的数据包的个数。在 avgidle 跌落到 0 之前，这么多的数据包可以突发传输出去。这个值越高，越能够容纳突发传输。你无法直接设置 maxidle 的值，必须通过这个参数来控制。

minburst

如前所述，发生越限时 CBQ 会禁止发包。实现这个的理想方案是根据事先计算出的闲置时间进行延迟之后，发一个数据包。然而，UNIX 的内核一般来说都有一个固定的调度周期(一般不大于 10ms)，所以最好是这样：禁止发包的时间稍长一些，然后突发性地传输 minburst 个数据包,而不是一个一个地传输。等待的时间叫做 offtime。

从大的时间尺度上说，minburst 值越大,整形越精确。但是，从毫秒级的时间尺度上说，就会有越多的突发传输。

minidle

如果 avgidle 值降到 0，也就是发生了越限，就需要等待，直到 avgidle 的值足够大才发送数据包。为避免因关闭链路太久而引起的以外突发传输，在 avgidle 的值太低的时候会被强制设置为 minidle 的值。

参数 minidle 的值是以负微秒记的。所以 10 代表 avgidle 被限制在-10us 上。

mpu

最小包尺寸——因为即使是 0 长度的数据包，在以太网上也要生成封装成 64 字节的帧，而需要一定时间去传输。为了精确计算闲置时间，CBQ 需要知道这个值。

rate

期望中的传输速率。也就是“油门”!

在 CBQ 的内部由很多的微调参数。比如，那些已知队列中没有数据的类就不参加计算、越限的类将被惩罚性地降低优先级等等。都非常巧妙和复杂。

9.5.4.2. CBQ 在分类方面的行为

除了使用上述 idletime 近似值进行整形之外，CBQ 还可以象 PRIO 队列那样，把

各种类赋予不同的优先级，优先权数值小的类会比优先权值大的类被优先处理。

每当网卡请求把数据包发送到网络上时，都会开始一个 WRR(weighted round robin，加权轮转)过程，从优先权值小的类开始。

那些队列中有数据的类就会被分组并被请求出队。在一个类收到允许若干字节数据出队的请求之后，再尝试下一个相同优先权值的类。

下面是控制 WRR 过程的一些参数：

allot

> 当从外部请求一个 CBQ 发包的时候，它就会按照“ priority ”参数指定的顺序轮流尝试其内部的每一个类的队列规定。当轮到一个类发数据时，它只能发送一定量的数据。“ allot ”参数就是这个量的基值。更多细节请参照“ weight ”参数。

prio

> CBQ 可以象 PRIO 设备那样工作。其中“ prio ”值较低的类只要有数据就必须先服务，其他类要延后处理。

weight

> “ weight ”参数控制 WRR 过程。每个类都轮流取得发包的机会。如果其中一个类要求的带宽显著地高于其他的类，就应该让它每次比其他的类发送更多的数据。
>
> CBQ 会把一个类下面所有的 weight 值加起来后归一化，所以数值可以任意定，只要保持比例合适就可以。人们常把“ 速率/10 ”作为参数的值来使用，实际工作得很好。归一化值后的值乘以“ allot ”参数后，决定了每次传输多少数据。

请注意，在一个 CBQ 内部所有的类都必须使用一致的主号码！

9.5.4.3. 决定链路的共享和借用的 CBQ 参数

除了纯粹地对某种数据流进行限速之外，CBQ 还可以指定哪些类可以向其它哪些类借用或者出借一部分带宽。

Isolated/sharing

> 凡是使用“ isolated ”选项配置的类，就不会向其兄弟类出借带宽。如果你的链路上同时存在着竞争对手或者不友好的其它人，你就可以使用这个选项。
>
> 选项“ sharing ”是“ isolated ”的反义选项。

bounded/borrow

> 一个类也可以用“ bounded ”选项配置，意味着它不会向其兄弟类借用带宽。选项“ borrow ”是“ bounded ”的反义选项。

一个典型的情况就是你的一个链路上有多个客户都设置成了“ isolated ”和“ bounded ”，那就是说他们都被限制在其要求的速率之下，且互相之间不会借用带宽。

在这样的一个类的内部的子类之间是可以互相借用带宽的。

9.5.4.4. 配置范例

这个配置把 WEB 服务器的流量控制为 5Mbps、SMTP 流量控制在 3Mbps 上。而且二者一共不得超过 6Mbps，互相之间允许借用带宽。我们的网卡是 100Mbps 的。

```
# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit         \
  avpkt 1000 cell 8
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit  \
  rate 6Mbit weight 0.6Mbit prio 8 allot 1514 cell 8 maxburst 20      \
  avpkt 1000 bounded
```

这部分按惯例设置了根为 1:0，并且绑定了类 1:1。也就是说整个带宽不能超过 6Mbps。

如前所述，CBQ 需要调整很多的参数。其实所有的参数上面都解释过了。相应的 HTB 配置则要简明得多。

```
# tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 100Mbit  \
  rate 5Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
# tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 100Mbit  \
  rate 3Mbit weight 0.3Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
```

我们建立了 2 个类。注意我们如何根据带宽来调整 weight 参数的。两个类都没有配置成“ bounded ”，但它们都连接到了类 1:1 上，而 1:1 设置了“ bounded ”。所以两个类的总带宽不会超过 6Mbps。别忘了，同一个 CBQ 下面的子类的主号码都必须与 CBQ 自己的号码相一致！

```
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
# tc qdisc add dev eth0 parent 1:4 handle 40: sfq
```

缺省情况下，两个类都有一个 FIFO 队列规定。但是我们把它换成 SFQ 队列，以保证每个数据流都公平对待。

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 80 0xffff flowid 1:3
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 25 0xffff flowid 1:4
```

这些命令规定了根上的过滤器，保证数据流被送到正确的队列规定中去。

注意：我们先使用了“ tc class add ” 在一个队列规定中创建了类，然后使用“ tc qdisc add ”在类中创建队列规定。

你可能想知道，那些没有被那两条规则分类的数据流怎样处理了呢？从这个例子来说，它们被 1:0 直接处理，没有限制。

如果 SMTP+web 的总带宽需求大于 6Mbps，那么这 6M 带宽将按照两个类的

weight 参数的比例情况进行分割：WEB 服务器得到 5/8 的带宽，SMTP 得到 3/8 的带宽。

从这个例子来说，你也可以这么认为：WEB 数据流总是会得到 5/8*6Mbps=3.75Mbps 的带宽。

9.5.4.5. 其它 CBQ 参数：split 和 defmap

如前所述，一个分类的队列规定需要调用过滤器来决定一个数据包应该发往哪个类去排队。

除了调用过滤器，CBQ 还提供了其他方式，defmap 和 split。很难掌握，但好在无关大局。但是现在是解释 defmap 和 split 的最佳时机，我会尽力解释。

因为你经常是仅仅需要根据 TOS 来进行分类，所以提供了一种特殊的语法。当 CBQ 需要决定了数据包要在哪里入队时，要检查这个节点是否为“ split 节点 ”。如果是，子队列规定中的一个应该指出它接收所有带有某种优先权值的数据包，权值可以来自 TOS 字段或者应用程序设置的套接字选项。

数据包的优先权位与 defmap 字段的值进行“ 或 ”运算来决定是否存在这样的匹配。换句话说，这是一个可以快捷创建仅仅匹配某种优先权值数据包的过滤器的方法。如果 defmap 等于 0xff，就会匹配所有包，0 则是不匹配。这个简单的配置可以帮助理解：

```
# tc qdisc add dev eth1 root handle 1: cbq bandwidth 10Mbit allot 1514 \
  cell 8 avpkt 1000 mpu 64

# tc class add dev eth1 parent 1:0 classid 1:1 cbq bandwidth 10Mbit    \
  rate 10Mbit allot 1514 cell 8 weight 1Mbit prio 8 maxburst 20        \
  avpkt 1000
```

一个标准的 CBQ 前导。

Defmap 参照 TC_PRIO 位(我从来不直接使用数字！)：

```
TC_PRIO..          Num  对应 TOS
-------------------------------------------------
BESTEFFORT         0    最高可靠性
FILLER             1    最低成本
BULK               2    最大吞吐量(0x8)
INTERACTIVE_BULK   4
INTERACTIVE        6    最小延迟(0x10)
CONTROL            7
```

TC_PRIO..的数值对应它右面的 bit。关于 TOS 位如何换算成优先权值的细节可以参照 pfifo_fast 有关章节。

然后是交互和大吞吐量的类：

```
# tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 10Mbit     \
  rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20        \
  avpkt 1000 split 1:0 defmap c0

# tc class add dev eth1 parent 1:1 classid 1:3 cbq bandwidth 10Mbit     \
  rate 8Mbit allot 1514 cell 8 weight 800Kbit prio 7 maxburst 20        \
  avpkt 1000 split 1:0 defmap 3f
```

“ split 队列规定 ”是 1:0，也就是做出选择的地方。c0 是二进制的 11000000，3F 是 00111111，所以它们共同匹配所有的数据包。第一个类匹配第 7 和第 6 位，也就是负责“ 交互 ”和“ 控制 ”的数据包。第二个类匹配其余的数据包。

节点 1:0 现在应该有了这样一个表格：

priority	send to
0	1:3
1	1:3
2	1:3
3	1:3
4	1:3
5	1:3
6	1:2
7	1:2

为了更有趣，你还可以传递一个“ change 掩码 ”，确切地指出你想改变哪个优先权值。你只有在使用了“ tc class change ”的时候才需要。比如，往 1:2 中添加 best effort 数据流，应该执行：

```
# tc class change dev eth1 classid 1:2 cbq defmap 01/01
```

现在，1:0 上的优先权分布应该是：

priority	send to
0	1:2
1	1:3
2	1:3
3	1:3
4	1:3
5	1:3
6	1:2
7	1:2

求助: 尚未测试过“ tc class change ”，资料上这么写的。

9.5.5. HTB(Hierarchical Token Bucket，分层的令牌桶)

Martin Devera (<devik>)正确地意识到 CBQ 太复杂，而且并没有按照多数常见情况进行优化。他的 Hierarchical 能够很好地满足这样一种情况：你有一个固定速率的链路，希望分割给多种不同的用途使用。为每种用途做出带宽承诺并实现定量的带宽借用。

HTB 就象 CBQ 一样工作，但是并不靠计算闲置时间来整形。它是一个分类的令牌桶过滤器。它只有很少的参数，并且在它的网站能够找到很好的文档。

随着你的 HTB 配置越来越复杂，你的配置工作也会变得复杂。但是使用 CBQ 的话，即使在很简单的情况下配置也会非常复杂！HTB3 (关于它的版本情况，请参阅它的网站)已经成了官方内核的一部分(2.4.20-pre1、2.5.31 及其后)。然而，你可能仍然要为你的 tc 命令打上 HTB3 支持补丁，否则你的 tc 命令不理解 HTB3。

如果你已经有了一个新版内核或者已经打了补丁，请尽量考虑使用 HTB。

9.5.5.1. 配置范例

环境与要求与上述 CBQ 的例子一样。

```
# tc qdisc add dev eth0 root handle 1: htb default 30

# tc class add dev eth0 parent 1: classid 1:1 htb rate 6mbit burst 15k

# tc class add dev eth0 parent 1:1 classid 1:10 htb rate 5mbit burst 15k
# tc class add dev eth0 parent 1:1 classid 1:20 htb rate 3mbit ceil 6mbit burst 15k
# tc class add dev eth0 parent 1:1 classid 1:30 htb rate 1kbit ceil 6mbit burst 15k
```

作者建议 2 在那些类的下方放置 SFQ：

```
# tc qdisc add dev eth0 parent 1:10 handle 10: sfq perturb 10
# tc qdisc add dev eth0 parent 1:20 handle 20: sfq perturb 10
# tc qdisc add dev eth0 parent 1:30 handle 30: sfq perturb 10
```

添加过滤器，直接把流量导向相应的类：

```
# U32="tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32"
# $U32 match ip dport 80 0xffff flowid 1:10
# $U32 match ip sport 25 0xffff flowid 1:20
```

这就完了——没有没见过的或者没解释过的数字，没有不明意义的参数。

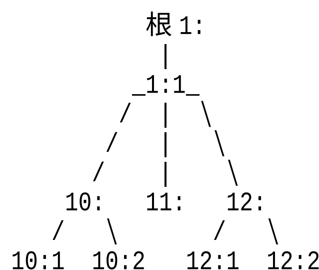
HTB 完成得相当不错——如果 10:和 20:都得到了保证的速率，剩下的就是分割了，它们借用的比率是 5:3，正如你其网的那样。

未被分类的流量被送到了 30:,仅有一点点带宽，但是却可以任意借用剩下的带宽。因为我们内部使用了 SFQ，而可以公平发包。

9.6. 使用过滤器对数据包进行分类

为了决定用哪个类处理数据包，必须调用所谓的“ 分类器链 ” 进行选择。这个链中包含了这个分类队列规定所需的所有过滤器。

重复前面那棵树：



当一个数据包入队的时候，每一个分支处都会咨询过滤器链如何进行下一步。典型的配置是在 1:1 处有一个过滤器把数据包交给 12:，然后 12:处的过滤器在把包交给 12:2。

你可以把后一个过滤器同时放在 1:1 处，可因为…having more specific tests lower in the chain.…而得到效率的提高。

另外，你不能用过滤器把数据包向“上”送。而且，使用 HTB 的时候应该把所有的规则放到根上！

再次强调：数据包只能向“下”进行入队操作！只有处队的时候才会上到网卡所在的位置来。他们不会落到树的最底层后送到网卡！

9.6.1. 过滤器的一些简单范例

就象在“分类器”那章所解释的，借助一些复杂的语法你可以详细地匹配任何事情。下面我们就开始，从简单地匹配一些比较明显的特征开始。

比方说，我们有一个 PRIO 队列规定，叫做“10:”，包含 3 个类，我们希望把去往 22 口的数据流发送到最优先的频道中去。应该这样设置过滤器：

```
# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 \
  match ip dport 22 0xffff flowid 10:1

# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 \
  match ip sport 80 0xffff flowid 10:1

# tc filter add dev eth0 protocol ip parent 10: prio 2 flowid 10:2
```

什么意思呢？是说：

向 eth0 上的 10:节点添加一个 u32 过滤规则，它的优先权是 1：凡是去往 22 口（精确匹配）的 IP 数据包，发送到频道 10:1。

向 eth0 上的 10:节点添加一个 u32 过滤规则，它的优先权是 1：凡是来自 80 口（精确匹配）的 IP 数据包，发送到频道 10:1。

向 eth0 上的 10:节点添加一个过滤规则，它的优先权是 2：凡是上面未匹配的 IP 数据包，发送到频道 10:2。

别忘了添加“dev eth0”（你的网卡或许叫别的名字），因为每个网卡的句柄都有完全相同的命名空间。

想通过 IP 地址进行筛选的话，这么敲：

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip dst 4.3.2.1/32 flowid 10:1

# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip src 1.2.3.4/32 flowid 10:1

# tc filter add dev eth0 protocol ip parent 10: prio 2      \
  flowid 10:2
```

这个例子把去往 4.3.2.1 和来自 1.2.3.4 的数据包送到了最高优先的队列，其它的则送到次高权限的队列。

你可以连续使用 match，想匹配来自 1.2.3.4 的 80 口的数据包的话，就这么敲：

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 match ip src 4.3.2.1/32 match ip
sport 80 0xffff flowid 10:1
```

9.6.2. 常用到的过滤命令一览

这里列出的绝大多数命令都根据这个命令改编而来：

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 ……
```

这些是所谓的“ u32 ”匹配，可以匹配数据包的任意部分。

根据源/目的地址

源地址段　'match ip src 1.2.3.0/24'

目的地址段　'match ip dst 4.3.2.0/24'

单个 IP 地址使用“ /32 ”作为掩码即可。

根据源/目的端口，所有 IP 协议

源　'match ip sport 80 0xffff'

目的 'match ip dport 80 0xffff'

根据 IP 协议 (tcp, udp, icmp, gre, ipsec)

使用/etc/protocols 所指定的数字。

比如： icmp 是 1：'match ip protocol 1 0xff'.

根据 fwmark

你可以使用 ipchains/iptables 给数据包做上标记，并且这个标记会在穿过网卡的路由过程中保留下来。如果你希望对来自 eth0 并从 eth1 发出的数据包做整形，这就很有用了。语法是这样的：

```
t
c filter add dev eth1 protocol ip parent 1:0 prio 1 handle 6 fw flowid 1:1
```

注意，这不是一个 u32 匹配！

你可以象这样给数据包打标记：

```
#
iptables -A PREROUTING -t mangle -i eth0 -j MARK --set-mark 6
```

数字 6 是可以任意指定的。

如果你不想去学习所有的 tc 语法，就可以与 iptables 结合，仅仅学习按 fwmark 匹配就行了。☺

按 TOS 字段

选择交互和最小延迟的数据流：

```
#
tc filter add dev ppp0 parent 1:0 protocol ip prio 10 u32 \
```

m

```
atch ip tos 0x10 0xff flowid 1:4
```

想匹配大量传输的话，使用“ 0x08 0xff ”。

关于更多的过滤命令，请参照“ 高级过滤 ” 那一章。

9.7. IMQ(Intermediate queueing device,中介队列设备)

中介队列设备不是一个队列规定,但它的使用与队列规定是紧密相连的。就 Linux 而言，队列规定是附带在网卡上的，所有在这个网卡上排队的数据都排进这个队列规定。根据这个概念，出现了两个局限：

1. 只能进行出口整形(虽然也存在入口队列规定，但在上面实现分类的队列规定的可能性非常小)。

2. 一个队列规定只能处理一块网卡的流量，无法设置全局的限速。

IMQ 就是用来解决上述两个局限的。简单地说，你可以往一个队列规定中放任何东西。被打了特定标记的数据包在 netfilter 的 NF_IP_PRE_ROUTING 和 NF_IP_POST_ROUTING 两个钩子函数处被拦截，并被送到一个队列规定中，该队列规定附加到一个 IMQ 设备上。对数据包打标记要用到 iptables 的一种处理方法。

这样你就可以对刚刚进入网卡的数据包打上标记进行入口整形，或者把网卡们当成一个个的类来看待而进行全局整形设置。你还可以做很多事情，比如：把 http 流量放到一个队列规定中去、把新的连接请求放到一个队列规定中去、……

9.7.1. 配置范例

我们首先想到的是进行入口整形，以便让你自己得到高保证的带宽☺。就象配置其它网卡一样：

```
tc qdisc add dev imq0 root handle 1: htb default 20

tc class add dev imq0 parent 1: classid 1:1 htb rate 2mbit burst 15k

tc class add dev imq0 parent 1:1 classid 1:10 htb rate 1mbit
tc class add dev imq0 parent 1:1 classid 1:20 htb rate 1mbit

tc qdisc add dev imq0 parent 1:10 handle 10: pfifo
tc qdisc add dev imq0 parent 1:20 handle 20: sfq

tc filter add dev imq0 parent 10:0 protocol ip prio 1 u32 match \
ip dst 10.0.0.230/32 flowid 1:10
```

在这个例子中，使用了 u32 进行分类。其它的分类器应该也能实现。然后，被打上标记的包被送到 imq0 排队。

```
iptables -t mangle -A PREROUTING -i eth0 -j IMQ --todev 0
```

```
ip link set imq0 up
```

iptables 的 IMQ 处理方法只能用在 PREROUTING 和 POSTROUTING 链的 mangle 表中。语法是：

```
IMQ [ --todev n ]
```

n: imq 设备的编号

注：ip6tables 也提供了这种处理方法。

请注意，如果数据流是事后才匹配到 IMQ 处理方法上的，数据就不会入队。数据流进入 imq 的确切位置取决于这个数据流究竟是流进的还是流出的。下面是 netfilter（也就是 iptables）在内核中预先定义优先级：

```
enum nf_ip_hook_priorities {
        NF_IP_PRI_FIRST = INT_MIN,
        NF_IP_PRI_CONNTRACK = -200,
        NF_IP_PRI_MANGLE = -150,
        NF_IP_PRI_NAT_DST = -100,
        NF_IP_PRI_FILTER = 0,
        NF_IP_PRI_NAT_SRC = 100,
        NF_IP_PRI_LAST = INT_MAX,
};
```

对于流入的包，imq 把自己注册为优先权等于 NF_IP_PRI_MANGLE+1，也就是说数据包在经过了 PREROUTING 链的 mangle 表之后才进入 imq 设备。

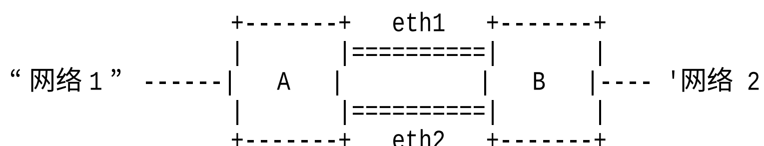
对于流出的包，imq 使用优先权等于 NF_IP_PRI_LAST，也就是说不会白白处理本应该被 filter 表丢弃的数据包。

关于补丁和更多的文档请参阅 imq 网站。

第 10 章 多网卡的负载均衡

有多种手段实现这个功能。最简单、最直接的方法之一就是“ TEQL ”——真（或“ 普通的 ”）链路均衡。就象用队列实现的大多数事情一样，负载均衡也需要双向实现。链路的两端都要参与，才有完整的效果。

想象下列情况：



A 和 B 是路由器，我们当然假定它们全是 Linux 机器。如果从网络 1 发往网络 2 的流量需要 A 路由器同时使用两条链路发给 B 路由器。B 路由器需要进行配置以便适应这种情况。反向传输时也一样，当数据包从网络 2 发往网络 1 时，B 路由器同时使用 eth1 和 eth2。

分配的功能是用“ TEQL ”设备实现的，象这样(没有比这更简单的了)：

```
# tc qdisc add dev eth1 root teql0
# tc qdisc add dev eth2 root teql0
# ip link set dev teql0 up
```

别忘了“ ip link set up ”命令！

这在两台机器上都要做。teql0 设备基本上是在 eth1 和 eth2 之间进行轮转发帧。用源也不会有数据从 teql 设备上进来，只是出现在原来的 eth1 和 eth2 上。

我们现在有了网络设备，还需要有合适的路由。方法之一就是给两个链路分配一个/31 的网络，teql0 也一样：

在 A 路由器上：

```
# ip addr add dev eth1 10.0.0.0/31
# ip addr add dev eth2 10.0.0.2/31
# ip addr add dev teql0 10.0.0.4/31
```

在 B 路由器上：

```
# ip addr add dev eth1 10.0.0.1/31
# ip addr add dev eth2 10.0.0.3/31
# ip addr add dev teql0 10.0.0.5/31
```

A 路由器现在应该能够通过 2 个真实链路和一个均衡网卡 ping 通 10.0.0.1、10.0.0.3 和 10.0.0.5。B 路由器应该能够 ping 通 10.0.0.0、10.0.0.2 和 10.0.0.4。

如果成功的话，A 路由器应该把 10.0.0.5 作为到达网络 2 的路由，B 路由器应该把 10.0.0.4 作为去往网络 1 的路由。在网络 1 是你家里的网络,而网络 2 是 Internet 这种特定场合下，A 路由器的缺省网关应该设为 10.0.0.5。

10.1. 告诫

事情永远不会是表面看上去那样简单。A 路由器和 B 路由器上的 eth1 和 eth2 需要关闭 “ 返回路径过滤 ”，否则它们会丢弃那些返回地址不同于其源地址的数据包：

```
# echo 0 > /proc/sys/net/ipv4/conf/eth1/rp_filter
# echo 0 > /proc/sys/net/ipv4/conf/eth2/rp_filter
```

包的乱序也是一个大问题。比如，有 6 个数据包需要从 A 发到 B，eth1 可能分到第 1、3、5 个包，而 eth2 分到第 2、4、6 个。在理想情况下，B 路由器会按顺序收到第 1、2、3、4、5、6 号包。但实际上 B 路由器的内核很可能按照类似 2、1、4、3、6、5 这样的随机顺序收到包。这个问题会把 TCP/IP 搞糊涂。虽然在链路上承载不同的 TCP/IP 会话并没有问题，但你无法通过捆绑多个链路来增加一个 ftp 文件的下载速度，除非两端的操作系统都是 Linux，因为 Linux 的 TCP/IP 协议栈不那么容易被这种简单的乱序问题所蒙蔽。

当然，对于大多数应用系统来说，链路的负载均衡是一个好主意。

10.2. 其它可能性

William Stearns 已经利用高级隧道来达到捆绑多重 Internet 连接的效果。可以在他的隧道网页找到。

本 HOWTO 将来可能更多地描述这个问题。

第 11 章 Netfilter 和 iproute——

迄今为止我们已经了解了 iproute 是如何工作的，并且多次提到了 netfilter。这里，你正好可以趁机看一看 Rusty 出名地不可靠的指南. Netfilter 本身可以在这里找到。

Netfilter 可以让我们进行包过滤或者篡改数据包头。有一个特别的功能就是我们可以给数据包打上一个数字标记。使用--set-mark 机制就可以。

例如，这个命令把所有发往 25/tcp（发出邮件）的数据包都打上了标记：

```
# iptables -A PREROUTING -i eth0 -t mangle -p tcp --dport 25 \
 -j MARK --set-mark 1
```

比如说，我们有多个连接，一个是按流量计费的，比较贵但是很快，另一个是包月的，但是比较慢。我们当然愿意让发送电子邮件的数据包走那条比较便宜的路由。

我们已经给那些数据包打上了标记“ 1 ”，我们现在命令路由策略数据库实现这个功能：

```
# echo 201 mail.out >> /etc/iproute2/rt_tables
# ip rule add fwmark 1 table mail.out
# ip rule ls
0:                                                                  f
rom all lookup local
32764:                                                              f
rom all fwmark        1 lookup mail.out
32766:                                                              f
rom all lookup main
32767:                                                              f
rom all lookup default
```

现在我们建立一个通往那条便宜链路的路由，从而生成 mail.out 路由表：

```
# /sbin/ip route add default via 195.96.98.253 dev ppp0 table mail.out
```

这就做完了。我们可能需要一些例外，有很多方法都能达到目的。我们可以修改 netfilter 命令来排除一些主机，也可以插入一些优先权值更低的规则把需要排除的主机的数据包发往 main 路由表。

我们还可以通过识别数据包的 TOS 位，来给不同服务类型的数据包打上不同的标记，再为它们分别建立规则。你甚至可以利用这种方法让诸如 ISDN 线路支持交互业务。

不用说，这当然也可以用于正在进行 NAT（“伪装”）的机器上。

重要提醒：我们收到报告说 MASQ 和 SNAT 功能与数据包标记有冲突。Rusty Russell 在这个帖子中作了解释。关闭反方向的过滤就可以正常工作。

注意：想给数据包打标记的话，你的内和需要一些配置：

```
IP: advanced router (CONFIG_IP_ADVANCED_ROUTER) [Y/n/?]
IP: policy routing (CONFIG_IP_MULTIPLE_TABLES) [Y/n/?]
IP: use netfilter MARK value as routing key (CONFIG_IP_ROUTE_FWMARK) [Y/n/?]
```

参考*方便菜谱一章中的* 15.5。

第 12 章 对包进行分类的高级过滤器

就象在分类的队列规定一段中解释的，过滤器用与把数据包分类并放入相应的子队列。这些过滤器在分类的队列规定内部被调用。

下面就是我们可用的分类器（部分）：

fw

根据防火墙如何对这个数据包做标记进行判断。如果你不想学习 tc 的过滤器语法，这倒是一个捷径。细节请参见队列那一章。

u32

根据数据包中的各个字段进行判断，如源 IP 地址等等。

route

根据数据包将被哪条路由进行路由来判断。

rsvp, rsvp6

根据数据包的 RSVP 情况进行判断。只能用于你自己的网络，互联网并不遵守 RSVP。

tcindex

用于 DSMARK 队列规定，参见相关章节。

通常来说，总有很多途径可实现对数据包分类，最终取决于你喜欢使用哪种系统。

分类器一般都能接受几个参数，为了方便我们列出来：

protocol

这个分类器所接受的协议。一般来说你只会接受 IP 数据。必要参数。

parent

这个分类器附带在哪个句柄上。句柄必须是一个已经存在的类。必要参数。

prio

这个分类器的优先权值。优先权值低的优先。

handle

对于不同过滤器，它的意义不同。

后面所有的节都假定你试图对去往 HostA 的流量进行整形。并且假定根类配置为 1:，并且你希望把选中的数据包送给 1:1 类。

12.1. u32 分类器

U32 分类器是当前实现中最先进的过滤器。全部基于哈希表实现，所以当有很多过滤器的时候仍然能够保持健壮。

U32 过滤器最简单的形式就是一系列记录，每条记录包含两个部分：一个选择器和一个动作。下面要讲的选择器用来与 IP 包相匹配，一旦成功匹配就执行其指定的动作。最简单的动作就是把数据包发送到特定的类队列。

用来配置过滤器的 tc 命令行由三部分组成：过滤器说明、选择器和动作。一个过滤器可以如下定义：

```
tc filter add dev IF [ protocol PROTO ]
                     [ (preference|priority) PRIO ]
                     [ parent CBQ ]
```

上面行中，protocol 字段描述了过滤器要匹配的协议。我们将只讨论 IP 协议的情况。preference 字段(也可以用 priority 代替)设置该过滤器的优先权。这非常重要，因为你可能有几条拥有不同优先权的过滤器。每个过滤器列表都按照输入的顺序被扫描一遍，然后优先权值低(更高的偏好值)的列表优先被处理。“ parent ”字段定义了过滤器所属的 CBQ 的顶部(如 1:0)。

上面描述的选项适用于所有过滤器，而不仅仅适用于 U32。

12.1.1. U32 选择器

u32 选择器包含了能够对当前通过的数据包进行匹配的特征定义。它其实只是定义了 IP 包头中某些位的匹配而已，但这种看似简单的方法却非常有效。让我们看看这个从实际应用的系统中抄来的例子：

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00100000 00ff0000 at 0 flowid 1:10
```

现在，命令的第一行已经不用解释了，前面都说过了。我们把精力集中在用“ match ”选项描述选择器的第二行。这个选择器将匹配那些 IP 头部的第二个字节是 0x10 的数据包。你应该猜到了，00ff 就是匹配掩码，确切地告诉过滤器应该匹配哪些位。在这个例子中是 0xff,所以会精确地匹配这个字节是否等于 0x10。“ at ”关键字的意思是指出从数据包的第几个字节开始匹配——本例中是从数据包的开头开始。完全地翻译成人类语言就是：“ 匹配那些 TOS 字段带有‘ 最小延迟 ’属性的数据包 ”。让我们看看另一个例子：

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00000016 0000ffff at nexthdr+0 flowid 1:10
```

“ nexthdr ”选项意味着封装在 IP 包中的下一个 PDU 的头部，也就是说它的上层协议的头。匹配操作就是从这个协议的头部开始的，应该发生在头部开始的第 16 位处。在 TCP 和 UDP 协议的头部，这个部分存放的是这个报文的目标端口。数字是按照先高厚低的格式存储的，所以 0x0016 就是十进制的 22(如果是 TCP 的话就是 ssh 服务)。其实，这个匹配在没有上下文的情况下含义很模糊，我们放

在后面讨论。

理解了上面的例子之后，下面这条选择器就很好懂了：

match c0a80100 ffffff00 at 16

表示了：匹配从 IP 头开始数的第 17 个字节到第 19 个字节。这个选择器将匹配所有去往 192.168.1.0/24 的数据包。成功分析完上面这个例子后，我们就已经掌握 u32 选择器了。

12.1.2. 普通选择器

普通选择器定义了要对数据包进行匹配的特征、掩码和偏移量。使用普通选择器，你实际上可以匹配 IP(或者上层协议)头部的任意一个 bit，虽然这样的选择器比特殊选择器难读和难写。一般选择器的语法是：

```
match [ u32 | u16 | u8 ] PATTERN MASK [ at OFFSET | nexthdr+OFFSET]
```

利用 u32、u16 或 u8 三个关键字中的一个来指明特征的 bit 数。然后 PATTERN 和 MASK 应该按照它定义的长度紧挨着写。OFFSET 参数是开始进行比较的偏移量(以字节计)。如果给出了“ nexthdr+ ”关键字，偏移量就移到上层协议头部开始的位置。

一些例子：

```
# tc filter add dev ppp14 parent 1:0 prio 10 u32 \
     match u8 64 0xff at 8 \
     flowid 1:4
```

如果一个数据包的 TTL 值等于 64，就将匹配这个选择器。TTL 就位于 IP 包头的第 9 个字节。

匹配带有 ACK 位的 TCP 数据包：

```
# tc filter add dev ppp14 parent 1:0 prio 10 u32 \
     match ip protocol 6 0xff \
     match u8 0x10 0xff at nexthdr+13 \
     flowid 1:3
```

用这个匹配小于 64 字节的 ACK 包：

```
## match acks the hard way,
## IP protocol 6,
## IP header length 0x5(32 bit words),
## IP Total length 0x34 (ACK + 12 bytes of TCP options)
## TCP ack set (bit 5, offset 33)
# tc filter add dev ppp14 parent 1:0 protocol ip prio 10 u32 \
            match ip protocol 6 0xff \
            match u8 0x05 0x0f at 0 \
            match u16 0x0000 0xffc0 at 2 \
            match u8 0x10 0xff at 33 \
            flowid 1:3
```

这个规则匹配了带有 ACK 位，且没有载荷的 TCP 数据包。这里我们看见了同时使用两个选择器的例子，这样用的结果是两个条件进行逻辑“ 与 ”运算。如果我们查查 TCP 头的结构，就会知道 ACK 标志位于第 14 个字节的第 5 个 bit(0x10)。

作为第二个选择器，如果我们采用更复杂的表达，可以写成“ match u8 0x06 0xff at 9 ”，而不是使用特殊选择器 protocol，因为 TCP 的协议号是 6(写在 IP 头的第十个字节)。另一方面，在这个例子中我们不使用特殊选择器也是因为没有用来匹配 TCP 的 ACK 标志的特殊选择器。

下面这个选择器是上面选择器的改进版，区别在于它不检查 IP 头部的长度。为什么呢？因为上面的过滤器只能在 32 位系统上工作。

```
tc filter add dev ppp14 parent 1:0 protocol ip prio 10 u32 \
     match ip protocol 6 0xff \
     match u8 0x10 0xff at nexthdr+13 \
     match u16 0x0000 0xffc0 at 2 \
     flowid 1:3
```

12.1.3. 特殊选择器

下面的表收入了本节文档的作者从 tc 程序的源代码中找出的所有特殊选择器。它们能够让你更容易、更可靠地配置过滤器。

求助: table placeholder - the table is in separate file ,,selector.html"

求助: it's also still in Polish :-(

求助: must be sgml'ized

一些范例：

```
# tc filter add dev ppp0 parent 1:0 prio 10 u32 \
     match ip tos 0x10 0xff \
     flowid 1:4
```

求助: tcp dport match does not work as described below:

上述规则匹配那些 TOS 字段等于 0x10 的数据包。TOS 字段位于数据包的第二个字节，所以与值等价的普通选择器就是：“ match u8 0x10 0xff at 1 ”。这其实给了我们一个关于 U32 过滤器的启示：特殊选择器全都可以翻译成等价的普通选择器，而且在内核的内存中，恰恰就是按这种方式存储的。这也可以导出另一个结论：tcp 和 udp 的选择器实际上是完全一样的，这也就是为什么不能仅用“ match tcp dport 53 0xffff ”一个选择器去匹配发到指定端口的 TCP 包，因为它也会匹配送往指定端口的 UDP 包。你一定不能忘了还得匹配协议类型，按下述方式来表示：

```
# tc filter add dev ppp0 parent 1:0 prio 10 u32 \
        match tcp dport 53 0xffff \
        match ip protocol 0x6 0xff \
        flowid 1:2
```

12.2. 路由分类器

这个分类器过滤器基于路由表的路由结果。当一个数据包穿越一个类，并到达一个标有“ route ”的过滤器的时候，它就会按照路由表内的信息进行分裂。当一个

数据包遍历类，并到达一个标记“ 路由 ”过滤器的时候，就会按照路由表的相应信息分类。

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 route
```

我们向节点 1:0 里添加了一个优先级是 100 的路由分类器。当数据包到达这个节点时，就会查询路由表，如果匹配就会被发送到给定的类，并赋予优先级 100。要最后完成，你还要添加一条适当的路由项：

这里的窍门就是基于目的或者源地址来定义“ realm ”。象这样做：

```
# ip route add Host/Network via Gateway dev Device realm RealmNumber
```

例如，我们可以把目标网络 192.168.10.0 定义为 realm 10：

```
# ip route add 192.168.10.0/24 via 192.168.10.1 dev eth1 realm 10
```

我们再使用路由过滤器的时候，就可以用 realm 号码来表示网络或者主机了，并可以用来描述路由如何匹配过滤器：

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route to 10 classid 1:10
```

这个规则说：凡是去往 192.168.10.0 子网的数据包匹配到类 1:10。

路由过滤器也可以用来匹配源策略路由。比如，一个 Linux 路由器的 eth2 上连接了一个子网：

```
# ip route add 192.168.2.0/24 dev eth2 realm 2
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route from 2 classid 1:2
```

这个规则说：凡是来自 192.168.2.0 子网(realm 2)的数据包，匹配到 1:2。

12.3. 管制分类器

为了能够实现更复杂的配置，你可以通过一些过滤器来匹配那些达到特定带宽的数据包。你可以声明一个过滤器来来按一定比率抑制传输速率，或者仅仅不匹配那些超过特定速率的数据包。

如果现在的流量是 5M，而你想把它管制在 4M，那么你要么可以停止匹配整个的 5M 带宽，要么停止匹配 1M 带宽，来给所配置的类进行 4M 速率的传输。

如果带宽超过了配置的速率，你可以丢包、可以重新分类或者看看是否别的过滤器能匹配它。

12.3.1. 管制的方式

有两种方法进行管制。

如果你编译内核的时候加上了“ Estimators ”，内核就可以替你为每一个过滤器测量通过了多少数据，多了还是少了。这些评估对于 CPU 来讲非常轻松，它只不过是每秒钟累计 25 次通过了多少数据，计算出速率。

另一种方法是在你的过滤器内部，通过 TBF(令牌桶过滤器)来实现。TBF 只匹配到达到你配置带宽的数据流，超过的部分则按照事先指定的“ 越限动作 ”来处理。

12.3.1.1. 靠内核评估

这种方式非常简单，只有一个参数 “ avrate ”。所有低于 avrate 的数据包被保留，并被过滤器分到所指定的类中去，而那些超过了 avrate 的数据包则按照越限动作来处理，缺省的越限动作是 “ reclassify ” (重分类)。

内核使用 EWMA 算法来核算带宽，以防止对瞬时突发过于敏感。

12.3.1.2. 靠令牌桶过滤器

使用下列参数：

- buffer/maxburst
- mtu/minburst
- mpu
- rate

它们的意义与前面介绍 TBF 时所说的完全一样。但仍然要指出的是：如果把一个 TBF 管制器的 mtu 参数设置过小的话，将没有数据包通过，whereas the egress TBF qdisc will just pass them slower。

另一个区别是，管制器只能够通过或者丢弃一个数据包，而不能因为为了延迟而暂停发送。

12.3.2. 越限动作

如果你的过滤器认定一个数据包越限，就会按照 “ 越限动作 ” 来处理它。当前，支持三种动作：

continue

　　让这个过滤器不要匹配这个包，但是其它的过滤器可能会匹配它。

drop

　　这是个非常强硬的选项——让越限的数据包消失。它的用途不多，经常被用于 ingress 管制。比如，你的 DNS 在请求流量大于 5Mbps 的时候就会失灵，你就可以利用这个来保证请求量不会超标。

Pass/OK

　　让数据包通过。可用于避免复杂的过滤器，但要放在合适的地方。

reclassify

最经常用于对数据包进行重分类以达到最好效果。这是缺省动作。

12.3.3. 范例

现在最真实的范例就是下面第十五章提到的 “ 防护 SYN 洪水攻击 ”。

求助: if you have used this, please share your experience with us

12.4. 当过滤器很多时如何使用散列表

如果你需要使用上千个规则——比如你有很多需要不同 QoS 的客户机——你可能会发现内核花了很多时间用于匹配那些规则。

缺省情况下，所有的过滤器都是靠一个链表来组织的，链表按 priority 的降序排列。如果你有 1000 个规则，那么就有可能需要 1000 次匹配来决定一个包如何处理。

而如果你有 256 个链表，每个链表 4 条规则的话，这个过程可以更快。也就是说如果你能把数据包放到合适的链表上，可能只需要匹配 4 次就可以了。

利用散列表可以实现。比如说你有 1024 个用户使用一个 Cable MODEM，IP 地址范围是 1.2.0.0 到 1.2.3.255，每个 IP 都需要不同容器来对待，比如 “ 轻量级 ”、“ 中量级 ” 和 “ 重量级 ”。你可能要写 1024 个规则，象这样：

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.1 classid 1:1
...
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.254 classid 1:3
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.255 classid 1:2
```

为了提高效率，我们应该利用 IP 地址的后半部分作为散列因子，建立 256 个散列表项。第一个表项里的规则应该是这样：

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.1.0 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.2.0 classid 1:3
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.0 classid 1:2
```

下一个表项应该这么开始：

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.1 classid 1:1
...
```

这样的话，最坏情况下也只需要 4 次匹配，平均 2 次。

具体配置有些复杂，但是如果你真有很多规则的话，还是值得的。

我们首先生成 root 过滤器，然后创建一个 256 项的散列表：

```
# tc filter add dev eth1 parent 1:0 prio 5 protocol ip u32
# tc filter add dev eth1 parent 1:0 prio 5 handle 2: protocol ip u32 divisor 256
```

然后我们向表项中添加一些规则：

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.0.123 flowid 1:1
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.1.123 flowid 1:2
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.3.123 flowid 1:3
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.4.123 flowid 1:2
```

这是第 123 项，包含了为 1.2.0.123、1.2.1.123、1.2.2.123 和 1.2.3.123 准备的匹配规则，分别把它们发给 1:1、1:2、1:3 和 1:2。注意，我们必须用 16 进制来表示散列表项，0x7b 就是 123。

然后创建一个 " 散列过滤器 "，直接把数据包发给散列表中的合适表项：

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 800:: \
        match ip src 1.2.0.0/16 \
        hashkey mask 0x000000ff at 12 \
        link 2:
```

好了，有些数字需要解释。我们定义散列表叫做 " 800: "，所有的过滤都从这里开始。然后我们选择源地址(它们位于 IP 头的第 12、13、14、15 字节)，并声明我们只对它的最后一部分感兴趣。这个例子中，我们发送到了前面创建的第 2 个散列表项。

这比较复杂，然而实际上确实有效而且性能令人惊讶。注意，这个例子我们也可以处理成理想情况——每个表项中只有一个过滤器！

第 13 章 内核网络参数

内核有很多可以在不同环境下调整的参数。通常，预设的缺省值可以满足 99% 的环境要求，we don't call this the Advanced HOWTO for the fun of it!

有个很有趣的地方：/proc/sys/net，你应该看看。这里一开始并没有把所有的内容归档，但我们正在尽力如此。

有时候你需要看看 Linux 的内核源代码、读读 Documentation/filesystems/proc.txt。绝大多数特性那里都有解释。

(求助)

13.1. 反向路径过滤

缺省情况下，路由器路由一切包，即使某些数据包“明显地”不属于你的网络。一个简单的例子就是私有 IP 空间溢出到了 Internet 上。如果你有一个网卡带有去往 195.96.96.0/24 的路由，你不会期望从这里收到来自 212.64.94.1 的数据包。

很多人都希望关掉这个功能，所以内核作者们按照这种要求做了。你可以利用 /proc 下的一些文件来配置内核是否使用这个功能。这种方法就叫“反向路径过滤”。基本上说，就是如果一个数据包的回应包不是从它来的网卡发出去，就认为这是一个伪造包，应该丢弃。

下面的片段将在所有的(包括将来的)网卡上启用这个功能：

```
# for i in /proc/sys/net/ipv4/conf/*/rp_filter ; do
>  echo 2 > $i
> done
```

依照上面的例子，如果一个源地址是来自 office+isp 子网的数据包，却从 Linux 路由器的 eth1 口到达，那么它将被丢弃。同理，如果一个来自 office 子网的数据包却声明它来自防火墙外面的某个地方，也会被丢弃。

上面说的是完整的反向路径过滤。却省情况是只基于直接与网卡相接的子网地址进行过滤。这是因为完整的反向路径过滤会破坏非对称路由(就是说数据包从一条路来而从另一条路离开——比如你在网络中使用了动态路由(bgp、ospf、rip)或者比如卫星通讯，数据从卫星下行到路由器，上行数据则经过地面线路传输。)。

如果你有这些情况，就可以简单地关掉卫星下行数据要进入那块网卡的 rp_filter。如果你想看一看丢弃了哪些包，可以通过相同目录下的 log_martians 文件通知内核向你的 syslog 中写日志。

```
# echo 1 >/proc/sys/net/ipv4/conf/<interfacename>/log_martians
```

求助: is setting the conf/[default,all]/* files enough? - martijn

13.2. 深层设置

有很多参数可以修改。我们希望能够全列出来。在 Documentation/ip-sysctl.txt 中也有部分记载。

这些设置中的部分缺省值取决于你在内核配置时是否选择了“ Configure as router and not host ”。

Oskar Andreasson 也有一个网页比我们讨论得更详细的网页：http://ipsysctl-tutorial.frozentux.net/

13.2.1. ipv4 一般设置

作为一个一般性的提醒，多数速度限制功能都不对 loopback 起作用，所以不要进行本地测试。限制是由“ jiffies ” 来提供的，并强迫使用前面提到过的 TBF。

内核内部有一个时钟，每秒钟发生“ HZ ”个 jiffies(滴嗒)。在 Intel 平台上，HZ 的值一般都是 100。所以设置*_rate 文件的时候，如果是 50，就意味着每秒允许 2 个包。TBF 配置成为如果有多余的令牌就允许 6 个包的突发。

下面列表中的一些条目摘录自 Alexey Kuznetsov kuznet@ms2.inr.ac.ru 和 Andi Kleen ak@muc.de 写的/usr/src/linux/Documentation/networking/ip-sysctl.txt。

/proc/sys/net/ipv4/icmp_destunreach_rate

> 一旦内核认为它无法发包，就会丢弃这个包，并向发包的主机发送 ICMP 通知。

/proc/sys/net/ipv4/icmp_echo_ignore_all

> 根本不要响应 echo 包。请不要设置为缺省，它可能在你正被利用成为 DoS 攻击的跳板时可能有用。

/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts [Useful]

> 如果你 ping 子网的子网地址，所有的机器都应该予以回应。这可能成为非常好用的拒绝服务攻击工具。设置为 1 来忽略这些子网广播消息。

/proc/sys/net/ipv4/icmp_echoreply_rate

> 设置了向任意主机回应 echo 请求的比率。

/proc/sys/net/ipv4/icmp_ignore_bogus_error_responses

> 设置它之后，可以忽略由网络中的那些声称回应地址是广播地址的主机生成的 ICMP 错误。

/proc/sys/net/ipv4/icmp_paramprob_rate

> 一个相对不很明确的 ICMP 消息，用来回应 IP 头或 TCP 头损坏的异常数据包。你可以通过这个文件控制消息的发送比率。

/proc/sys/net/ipv4/icmp_timeexceed_rate

这个在 traceroute 时导致著名的“ Solaris middle star ”。这个文件控制发送 ICMP Time Exceeded 消息的比率。

/proc/sys/net/ipv4/igmp_max_memberships

主机上最多有多少个 igmp (多播)套接字进行监听。

求助: Is this true?

/proc/sys/net/ipv4/inet_peer_gc_maxtime

求助: Add a little explanation about the inet peer storage? Minimum interval between garbage collection passes. This interval is in effect under low (or absent) memory pressure on the pool. Measured in jiffies.

/proc/sys/net/ipv4/inet_peer_gc_mintime

每一遍碎片收集之间的最小时间间隔。当内存压力比较大的时候，调整这个间隔很有效。以 jiffies 计。

/proc/sys/net/ipv4/inet_peer_maxttl

entries 的最大生存期。在 pool 没有内存压力的情况下(比如，pool 中 entries 的数量很少的时候)，未使用的 entries 经过一段时间就会过期。以 jiffies 计。

/proc/sys/net/ipv4/inet_peer_minttl

entries 的最小生存期。应该不小于汇聚端分片的生存期。当 pool 的大小不大于 inet_peer_threshold 时，这个最小生存期必须予以保证。以 jiffies 计。

/proc/sys/net/ipv4/inet_peer_threshold

The approximate size of the INET peer storage. Starting from this threshold entries will be thrown aggressively. This threshold also determines entries' time-to-live and time intervals between garbage collection passes. More entries, less time-to-live, less GC interval.

/proc/sys/net/ipv4/ip_autoconfig

这个文件里面写着一个数字，表示主机是否通过 RARP、BOOTP、DHCP 或者其它机制取得其 IP 配置。否则就是 0。

/proc/sys/net/ipv4/ip_default_ttl

数据包的生存期。设置为 64 是安全的。如果你的网络规模巨大就提高这个值。不要因为好玩而这么做——那样会产生有害的路由环路。实际上，在很多情况下你要考虑能否减小这个值。

/proc/sys/net/ipv4/ip_dynaddr

如果你有一个动态地址的自动拨号接口，就得设置它。当你的自动拨号接口激活的时候，本地所有没有收到答复的 TCP 套接字会重新绑定到正确的地址上。这可以解决引发拨号的套接字本身无法工作，重试一次却可以的问题。

/proc/sys/net/ipv4/ip_forward

内核是否转发数据包。缺省禁止。

/proc/sys/net/ipv4/ip_local_port_range

用于向外连接的端口范围。缺省情况下其实很小：1024 到 4999。

/proc/sys/net/ipv4/ip_no_pmtu_disc

如果你想禁止"沿途 MTU 发现"就设置它。"沿途 MTU 发现"是一种技术，可以在传输路径上检测出最大可能的 MTU 值。参见 *Cookbook* 一章中关于"沿途 MTU 发现"的内容。

/proc/sys/net/ipv4/ipfrag_high_thresh

用于 IP 分片汇聚的最大内存用量。分配了这么多字节的内存后，一旦用尽，分片处理程序就会丢弃分片。When ipfrag_high_thresh bytes of memory is allocated for this purpose, the fragment handler will toss packets until ipfrag_low_thresh is reached.

/proc/sys/net/ipv4/ip_nonlocal_bind

如果你希望你的应用程序能够绑定到不属于本地网卡的地址上时，设置这个选项。如果你的机器没有专线连接(甚至是动态连接)时非常有用，即使你的连接断开，你的服务也可以启动并绑定在一个指定的地址上。

/proc/sys/net/ipv4/ipfrag_low_thresh

用于 IP 分片汇聚的最小内存用量。

/proc/sys/net/ipv4/ipfrag_time

IP 分片在内存中的保留时间(秒数)。

/proc/sys/net/ipv4/tcp_abort_on_overflow

一个布尔类型的标志，控制着当有很多的连接请求时内核的行为。启用的话，如果服务超载，内核将主动地发送 RST 包。

/proc/sys/net/ipv4/tcp_fin_timeout

如果套接字由本端要求关闭，这个参数决定了它保持在 FIN-WAIT-2 状态的时间。对端可以出错并永远不关闭连接，甚至意外当机。缺省值是 60 秒。2.2 内核的通常值是 180 秒，你可以按这个设置，但要记住的是，即使你的机器是一个轻载的 WEB 服务器，也有因为大量的死套接字而内存溢出的风险，FIN-WAIT-2 的危险性比 FIN-WAIT-1 要小，因为它最多只能吃掉 1.5K 内存，但是它们的生存期长些。参见 tcp_max_orphans。

/proc/sys/net/ipv4/tcp_keepalive_time

当 keepalive 起用的时候，TCP 发送 keepalive 消息的频度。缺省是 2 小时。

/proc/sys/net/ipv4/tcp_keepalive_intvl

当探测没有确认时，重新发送探测的频度。缺省是 75 秒。

/proc/sys/net/ipv4/tcp_keepalive_probes

在认定连接失效之前，发送多少个 TCP 的 keepalive 探测包。缺省值是 9。这个值乘以 tcp_keepalive_intvl 之后决定了，一个连接发送了 keepalive 之后可以有多少时间没有回应。

/proc/sys/net/ipv4/tcp_max_orphans

系统中最多有多少个 TCP 套接字不被关联到任何一个用户文件句柄上。如果超过这个数字，孤儿连接将即刻被复位并打印出警告信息。这个限制仅仅是为了防止简单的 DoS 攻击，你绝对不能过分依靠它或者人为地减小这个值，更应该增加这个值(如果增加了内存之后)。This limit exists only to prevent simple DoS attacks, you _must_ not rely on this or lower the limit artificially, but rather increase it (probably, after increasing installed memory), if network conditions require more than default value, and tune network services to linger and kill such states more aggressively. 让我再次提醒你：每个孤儿套接字最多能够吃掉你 64K 不可交换的内存。

/proc/sys/net/ipv4/tcp_orphan_retries

本端试图关闭 TCP 连接之前重试多少次。缺省值是 7，相当于 50 秒~16 分钟(取决于 RTO)。如果你的机器是一个重载的 WEB 服务器，你应该考虑减低这个值，因为这样的套接字会消耗很多重要的资源。参见 tcp_max_orphans。

/proc/sys/net/ipv4/tcp_max_syn_backlog

记录的那些尚未收到客户端确认信息的连接请求的最大值。对于有 128M 内存的系统而言，缺省值是 1024，小内存的系统则是 128。如果服务器不堪重负，试试提高这个值。注意！如果你设置这个值大于 1024，最好同时调整 include/net/tcp.h 中的 TCP_SYNQ_HSIZE，以保证 TCP_SYNQ_HSIZE*16 ≤tcp_max_syn_backlo，然后重新编译内核。

/proc/sys/net/ipv4/tcp_max_tw_buckets

系统同时保持 timewait 套接字的最大数量。如果超过这个数字，time-wait 套接字将立刻被清除并打印警告信息。这个限制仅仅是为了防止简单的 DoS 攻击，你绝对不能过分依靠它或者人为地减小这个值，如果网络实际需要大于缺省值，更应该增加这个值(如果增加了内存之后)。

/proc/sys/net/ipv4/tcp_retrans_collapse

为兼容某些糟糕的打印机设置的“将错就错”选项。再次发送时，把数据包增大一些，来避免某些 TCP 协议栈的 BUG。

/proc/sys/net/ipv4/tcp_retries1

在认定出错并向网络层提交错误报告之前，重试多少次。缺省设置为 RFC 规定的最小值：3，相当于 3 秒~8 分钟（取决于 RIO）。

/proc/sys/net/ipv4/tcp_retries2

在杀死一个活动的 TCP 连接之前重试多少次。RFC 1122 规定这个限制应该长于 100 秒。这个值太小了。缺省值是 15，相当于 13~30 分钟（取决于 RIO）。

/proc/sys/net/ipv4/tcp_rfc1337

这个开关可以启动对于在 RFC1337 中描述的 " tcp 的 time-wait 暗杀危机 " 问题的修复。启用后，内核将丢弃那些发往 time-wait 状态 TCP 套接字的 RST 包。却省为 0。

/proc/sys/net/ipv4/tcp_sack

特别针对丢失的数据包使用选择性 ACK，这样有助于快速恢复。

/proc/sys/net/ipv4/tcp_stdurg

使用 TCP 紧急指针的主机需求解释。因为绝大多数主机采用 BSD 解释，所以如果你在 Linux 上打开它，可能会影响它与其它机器的正常通讯。缺省是 FALSE。

/proc/sys/net/ipv4/tcp_syn_retries

在内核放弃建立连接之前发送 SYN 包的数量。

/proc/sys/net/ipv4/tcp_synack_retries

为了打开对端的连接，内核需要发送一个 SYN 并附带一个回应前面一个 SYN 的 ACK。也就是所谓三次握手中的第二次握手。这个设置决定了内核放弃连接之前发送 SYN+ACK 包的数量。

/proc/sys/net/ipv4/tcp_timestamps

时间戳可以避免序列号的卷绕。一个 1Gbps 的链路肯定会遇到以前用过的序列号。时间戳能够让内核接受这种 " 异常 " 的数据包。

/proc/sys/net/ipv4/tcp_tw_recycle

能够更快地回收 TIME-WAIT 套接字。缺省值是 1。除非有技术专家的建议和要求，否则不应修改。

/proc/sys/net/ipv4/tcp_window_scaling

一般来说 TCP/IP 允许窗口尺寸达到 65535 字节。对于速度确实很高的网络而言这个值可能还是太小。这个选项允许设置上 G 字节的窗口大小，有利于在带宽*延迟很大的环境中使用。

13.2.2. 网卡的分别设置

DEV 可以是真实网卡名，或者是“ all ”、“ default ”。Defaulr 同时将来才出现的网卡有效。

/proc/sys/net/ipv4/conf/DEV/accept_redirects

如果路由器认定你使用不当(比如，它发现要在收到包的网卡上转发出数据包)，它就会发送 ICMP 重定向报文。然而这会引发一些轻微的安全问题，所以你可能希望关掉它，或者使用安全的重定向。

/proc/sys/net/ipv4/conf/DEV/accept_source_route

已经不是那么常用了。以前，你可以发出一个数据包，其中指出它应该路经的路由器的 IP。Linux 可以设置为尊重这个 IP 选项。

/proc/sys/net/ipv4/conf/DEV/bootp_relay

接受来自 0.b.c.d，去往非本地地址的数据包。并假定 BOOTP 中继守护程序会得到并转发这个包。

缺省值为 0。至少到 2.2.12 为止这个功能还没有完成。

/proc/sys/net/ipv4/conf/DEV/forwarding

在该网卡上启用或禁止 IP 转发。

/proc/sys/net/ipv4/conf/DEV/log_martians

参见 *Reverse Path Filtering*.

/proc/sys/net/ipv4/conf/DEV/mc_forwarding

是否在这个网卡上进行多播转发。

/proc/sys/net/ipv4/conf/DEV/proxy_arp

如果设置为 1，这个网卡将回应那些询问内核已知路由的 IP 地址的 ARP 请求。这对于构建“ ip 伪网桥 ”非常有用。在启用它之前，一定要确认你的子网掩码正确！还要保证前面提到的 rp_filter 能够正确处理 ARP 请求！

参见 16.3 用 ARP 代理实现伪网桥。

/proc/sys/net/ipv4/conf/DEV/rp_filter

参见 *Reverse Path Filtering*.

/proc/sys/net/ipv4/conf/DEV/secure_redirects

只接受对于网关的 ICMP 重定向消息，网关写在缺省网关列表中。缺省打开。

/proc/sys/net/ipv4/conf/DEV/send_redirects

我们是否发送上述重定向。

/proc/sys/net/ipv4/conf/DEV/shared_media

告诉内核这个设备上是否有能够直接访问到的不同子网。缺省是“ yes ”。

/proc/sys/net/ipv4/conf/DEV/tag

求助: fill this in

13.2.3. 邻居策略

Dev 可以是标准的真实网卡，也可以是“ all ”或者“ default ” 。default 也会修改将要生成的网卡的设置。

/proc/sys/net/ipv4/neigh/DEV/anycast_delay

回答邻居请求消息前的随机延迟的最大值，以 jiffies 计。尚未完成 (Linux 还不支持 anycast)。

/proc/sys/net/ipv4/neigh/DEV/app_solicit

决定了发往用户级 ARP 守护程序的请求数量。设为 0 等于关闭。

/proc/sys/net/ipv4/neigh/DEV/base_reachable_time

一个根据 RFC2461 来计算随机可到达时间的基值。

/proc/sys/net/ipv4/neigh/DEV/delay_first_probe_time

第一次探测邻机是否可到达的延迟。(参见 gc_stale_time)

/proc/sys/net/ipv4/neigh/DEV/gc_stale_time

决定了检查陈旧 ARP 条目的频度。当一个 ARP 条目老化之后将被重新解析(尤其是一个 IP 地址转移到另一台机器上的时候)。如果 ucast_solicit 值大于 0，就会先试图向已知主机发送一个单播 ARP 包，如果失败并且 mcast_solicit 值大于 0 的话，在发送广播 ARP 包。

/proc/sys/net/ipv4/neigh/DEV/locktime

如果一个 ARP/neighbor 条目超过了 locktime，就将被一个新的条目取代。这可以防止 ARP 缓冲过于臃肿。

/proc/sys/net/ipv4/neigh/DEV/mcast_solicit

多播请求的最大重试次数。

/proc/sys/net/ipv4/neigh/DEV/proxy_delay

在启用了 ARP 代理的网卡上，回应一个 ARP 请求前需要等待的时间(实际值是[0..proxytime]区间的随机值)。在某些情况下，这可以避免网络洪水。

/proc/sys/net/ipv4/neigh/DEV/proxy_qlen

用于延迟代理 ARP 定时器的最大队列长度(参见 proxy_delay)。

/proc/sys/net/ipv4/neigh/DEV/retrans_time

两次重发 Neighbor Solicitation 消息之间的时间间隔，以 jiffies 计(1/100 秒)。用于地址解析和确定邻居是否可到达。

/proc/sys/net/ipv4/neigh/DEV/ucast_solicit

单播请求的最大次数。

/proc/sys/net/ipv4/neigh/DEV/unres_qlen

存储未完成 ARP 请求的最大队列长度(包的个数)。未完成 ARP 请求：be accepted from other layers while the ARP address is still resolved.

Internet QoS: Architectures and Mechanisms for Quality of Service, Zheng Wang, ISBN 1-55860-608-4

一本精装教科书，涵盖了与 QoS 相关的主题。对于掌握基本概念很有好处。

13.2.4. 路由设置

/proc/sys/net/ipv4/route/error_burst

这些参数用来限制从路由代码写入内核日志的 warning 消息。error_cost 的值越高，写的消息越少。error_burst 控制什么时候应该丢弃消息。缺省设置是限制为每 5 秒钟一次。

/proc/sys/net/ipv4/route/error_cost

这些参数用来限制从路由代码写入内核日志的 warning 消息。error_cost 的值越高，写的消息越少。error_burst 控制什么时候应该丢弃消息。缺省设置是限制为每 5 秒钟一次。

/proc/sys/net/ipv4/route/flush

对这个文件的写操作导致刷新路由缓冲。

/proc/sys/net/ipv4/route/gc_elasticity

一个数值，用来控制路由缓冲碎片收集算法的频度和行为。用来设置失败恢复时非常重要。一旦上一次路由操作失败，经过这么多秒后，Linux 会跳过失败而转向其它路由。缺省值为 300，如果你想要更快速的失败恢复就可以降低这个值。

参见 Ard van Breemen 的这篇帖子。

/proc/sys/net/ipv4/route/gc_interval

参见 /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/gc_min_interval

参见 /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/gc_thresh

参见 /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/gc_timeout

参见 /proc/sys/net/ipv4/route/gc_elasticity.

/proc/sys/net/ipv4/route/max_delay

刷新路由缓冲的延迟。

/proc/sys/net/ipv4/route/max_size

路由缓冲的最大值。一旦缓冲满，就清除旧的条目。

/proc/sys/net/ipv4/route/min_adv_mss

求助: fill this in

/proc/sys/net/ipv4/route/min_delay

刷新路由缓冲的延迟。

/proc/sys/net/ipv4/route/min_pmtu

求助: fill this in

/proc/sys/net/ipv4/route/mtu_expires

求助: fill this in

/proc/sys/net/ipv4/route/redirect_load

决定是否要向特定主机发送更多的 ICMP 重定向的因子。一旦达到 load 限制或者 number 限制就不再发送。

/proc/sys/net/ipv4/route/redirect_number

参见 /proc/sys/net/ipv4/route/redirect_load.

/proc/sys/net/ipv4/route/redirect_silence

重定向的超时。经过这么长时间后，重定向会重发，而不管是否已经因为超过 load 或者 number 限制而停止。

第 14 章 不经常使用的高级队列规定

你应该发现有时候前面提到的那些队列不能满足你的需要，这里列出了一些内核包含的其它更多类型的队列。

14.1. bfifo/pfifo

这些无类的队列因为没有内部频道而显得比 pfifo_fast 还要简单，所有的流量都均等处理。但是它们还是有一个好处：他们可以做统计。所以即使你不想整形或排序，你也可是使用这个队列规定来检查网卡的后台日志。

pfifo 的长度单位以包的个数计，而 bfifo 以字节计。

14.1.1. 参数与使用

limit

> 规定了队列的长度。对于 bfifo 用字节计，对于 pfifo 用包的个数计。缺省值就是网卡的 txqueuelen 个包那么长(参见 pfifo_fast 那一章)，对于 bfifo 就是 txqueuelen*mtu 个字节。

14.2. Clark-Shenker-Zhang 算法 (CSZ)

它的理论性是如此之强，以至于连 Alexey(CBQ 的主要作者)都不指望去理解它。从它的原始资料来看：

> David D. Clark、Scott Shenker 和 Lixia Zhang *在综合业务分组网络中支持实时应用系统：体系与机制。*
>
> 据我的理解，它的主要意图是为每一个担保的服务创建 WFQ 流，并分配剩下的带宽给伪设备 flow-0。Flow-0 由服务预测和 best effort traffic 组成，并由一个优先调度器来处理……（谁能帮忙翻一下?☹）。As I understand it, the main idea is to create WFQ flows for each guaranteed service and to allocate the rest of bandwith to dummy flow-0. Flow-0 comprises the predictive services and the best effort traffic; it is handled by a priority scheduler with the highest priority band allocated for predictive services, and the rest --- to the best effort packets.
>
> Note that in CSZ flows are NOT limited to their bandwidth. It is

supposed that the flow passed admission control at the edge of the QoS network and it doesn't need further shaping. Any attempt to improve the flow or to shape it to a token bucket at intermediate hops will introduce undesired delays and raise jitter.

迄今为止，CSZ 是唯一一个能够真正提供服务担保的调度器。其它的方案(包括 CBQ)不提供延迟保证和随机抖动。"

现在看来不是一个好的候选方案，除非你已经阅读并理解了上面提到的文章。

14.3. DSMARK

Esteve Camps <marvin@grn.es>

这些文字是从我 2000 年 9 月的《*Linux 的 QoS*》*支持*这篇论文上摘录的。

参考文档：

- Draft-almesberger-wajhak-diffserv-linux-01.txt.
- iproute2 发行版中的范例。
- *QOS 论坛*的 QoS 协议与体系白皮书和 IP QoS 常见问题。

本章作者：Esteve Camps <esteve@hades.udg.es>.

14.3.1. 介绍

首先的首先，你应该到 IETF DiffServ 工作组的网站和 Werner Almesberger 的网站(他为 Linux 的 Differentiated Services 撰写了代码)阅读一下有关的 RFC(RFC2474、RFC2475、RFC2597 和 RFC2598)。.

14.3.2. Dsmark 与什么相关？

Dsmark 是一个队列规定，提供了 Differentiated Services(也叫 DiffServ，简称为 DS)所需要的能力。DiffServ 是两种 QoS 体系中的一种(另一种叫做 Integrated Services)，基于 IP 头中 DS 字段的值来工作。

最开始的实现方案中的一种是 IP 头中的服务类型字段(TOS 值)，也就是设计由 IP 来提供一些 QoS 级别。改变这个值，我们可以选择吞吐量、延迟或者可靠性的高/低级别。但是这并没有提供足够的灵活性来应付新的服务(比如说实时应用、交互应用等等)。后来，新的体系出现了。其中之一就是 DiffServ，保留了 TOS 位并 renamed DS 字段。

14.3.3. Differentiated Services 指导

Differentiated Services 是面向组的。也就是说，我们没有必要了解数据流(那是 Integrated Services 的风格)，我们只了解 “ 流会聚 ”，并根据一个数据包属于哪个流会聚来采取不同的行动。

当一个数据包到达边缘节点(entry node to a DiffServ domain)，并进入 DiffServ 域之后，我们就得进行对它们进行策略控制、整形或标记(关于标记请参考 “ 给 DS 字段分配一个值 ”。就象一头牛一样。☺)。这个值将被 DiffServ 域中的内部/核心代码参考，以决定如何处置或者适用什么 QoS 级别。

如你的推断，DS 包括一个 “ 域 ” 的概念，他规定了所有 DS 规则的作用范围。事实上你可以认为我把所有的数据包都分类到我的域中。他们一旦进入我的域，就会根据分类而隶属于某条规则，每个所经过的节点都会应用那个 QoS 级别。

其实你可以在你的本地域内使用你自己的策略控制器，但当与其它的 DS 域相连时要考虑一些*服务级别协议*。

这里，你可能由很多问题。DiffServ 不仅限于我所解释的。其实你应该能理解我不可能在 50 行中引用 3 个 RFC。

14.3.4. 使用 Dsmark

根据 DiffServ 文档的说法，我们区分边界节点和内部节点。这是传输路径上的两个关键点。但数据包到达时二者都会对包进行分类。在数据包被发送到网络上之前，它的分类结果可能会被这个 DS 过程的其他地方用到。正是因此，Diffserv 的代码提供了一个叫做 sk_buff 的结构，包含了一个新的字段叫做 skb->tc_index 用来存储一开始分类的结果，以便其它可能用到这个结果的地方引用。

skb->tc_index 的值被 DSMARK 队列规定根据每个 IP 包头的 DS 字段的值进行初始设置。此外，cls_tcindex 分类器将读取 skb->tcindex 的全部或部分，并用来选择类。

但是，首先看一看 DSMARK 队列规定的命令行和参数：

```
... dsmark indices INDICES [ default_index DEFAULT_INDEX ] [ set_tc_index ]
```

这些参数是什么意思呢？

- *indices*：存储(掩码，数值)的表格尺寸。最大值是 2ˆ n (n≥0)。
- *Default_index*: 当分类器没有找到匹配项时的缺省表格项索引。
- *Set_tc_index*: 给 dsmark 设置接收 DS 字段并存储到 skb->tc_index 规定。

让我们看看 DSMARK 的工作过程。

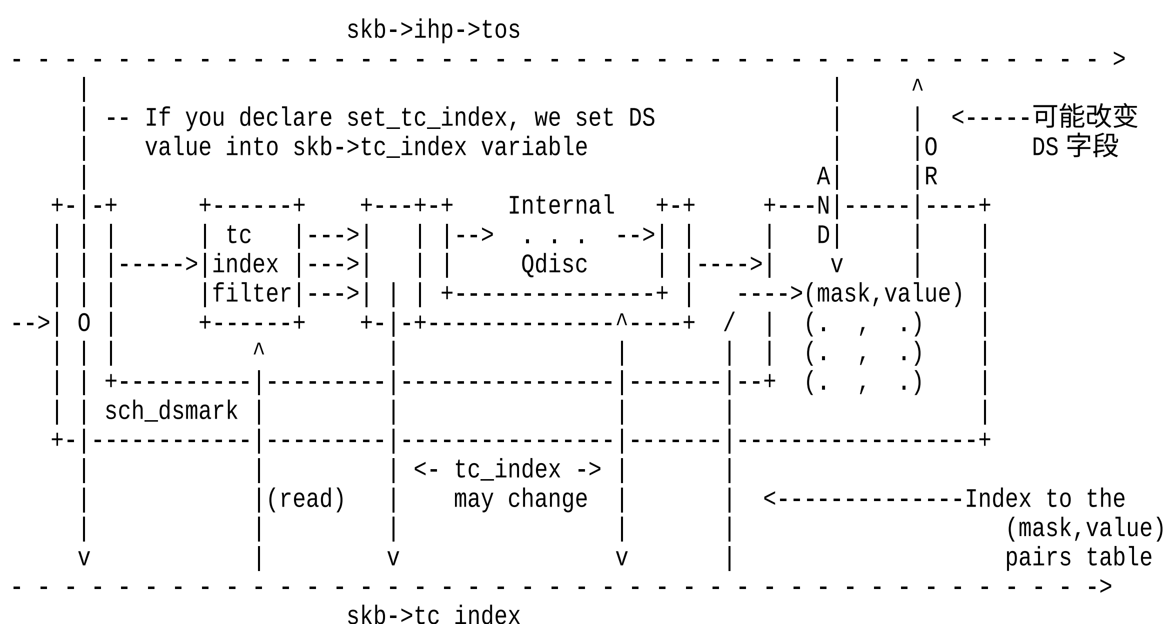
14.3.5. SCH_DSMARK 如何工作

这个队列规定将进行下列步骤：

- 如果我们在队列规定的命令行中声明了 set_tc_index 选项，DS 字段的值将被取出并存储在 skb->tc_index 中。
- 调用分类器。分类器被调用并将返回将被存储于 skb_tcindex 的类编号。如果没有找到能匹配的过滤器，会返回 default_index 的值。如果既没有声明 set_tc_index 又没有声明 default_index 的值，结果可能是不可预知的。
- 在数据包被发送到你可以重用分类器的结果的内部队列之后，由内部队列规定返回的类代码被保存在 skb->tc_index。我们将来检索 mask- value 表的时候还会用到这个值。随后的结果将决定数据包下一步的处理：

```
New_Ds_field = ( Old_DS_field & mask ) | value
```

- 然后，值将与 ds_field 的值和掩码进行“ 与 ”运算，然后再与参数值相“ 或 ”。下面的图表有利于你理解这个过程：



如何进行标记？只需改变你想重新标记的类的掩码和值就行了。看着一行代码：

```
tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
```

它改变了散列表中的(mask,value)记录，来重新标记属于类 1:1 的数据包。你必须使用 “ change ” 命令，因为(mask,value)记录已经有了缺省值。(see table below).

现在，我们解释一下 TC_INDEX 过滤器如何工作和适用场合。此外，TC_INDEX 过滤器也可以应用于没有 DS 服务的配置下。

14.3.6. TC_INDEX 过滤器

这是声明 TC_INDEX 过滤器的基本命令：

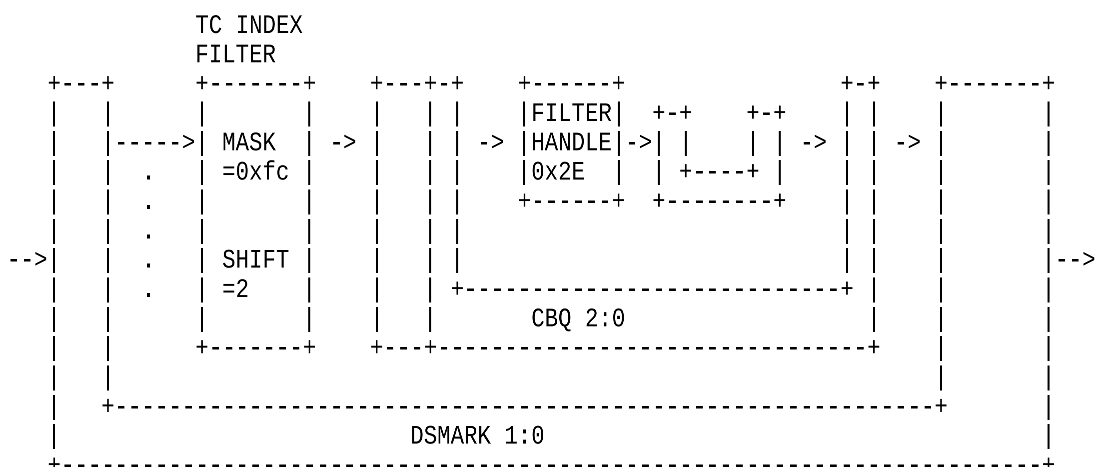
```
... tcindex [ hash SIZE ] [ mask MASK ] [ shift SHIFT ]
            [ pass_on | fall_through ]
            [ classid CLASSID ] [ police POLICE_SPEC ]
```

接下来，我们说明一下用来解释 TC_INDEX 操作模式的例子。注意斜体字：

```
tc qdisc add dev eth0 handle 1:0 root dsmark indices 64 set_tc_index
tc filter add dev eth0 parent 1:0 protocol ip prio 1 tcindex mask 0xfc shift 2
tc qdisc add dev eth0 parent 1:0 handle 2:0 cbq bandwidth 10Mbit cell 8 avpkt 1000 mpu 64 # EF
traffic class
tc class add dev eth0 parent 2:0 classid 2:1 cbq bandwidth 10Mbit rate 1500Kbit avpkt 1000 prio
1 bounded isolated allot 1514 weight 1 maxburst 10 # Packet fifo qdisc for EF traffic
tc qdisc add dev eth0 parent 2:1 pfifo limit 5 tc filter add dev eth0 parent 2:0 protocol ip
prio 1 handle 0x2e tcindex classid 2:1 pass_on
```

(这里的代码尚未完成。只是从 iproute2 发行中的 EFCBQ 范例中摘出来的。)。

首先，假设我们收到标记为“ EF ”的数据包。如果你读了 RFC2598，你就会明白 DSCP 推荐 EF 数据包的值是 101110。也就是说 DS 字段的值是 10111000(记住 TOS 字节中次要的位没有在 DS 中使用)或者是 16 进制的 0xb8。



数据包到来后就把 DS 字段设置为 0xb8。象前面所解释的，例子中被标记为 1:0 的 dsmark 队列规定找到 DS 字段的值并存储在 skb->tc_index 变量中。下一步将对应关联到这个队列规定的过滤器(例子中的第二行)。这将进行下列操作：

```
Value1 = skb->tc_index & MASK
Key = Value1 >> SHIFT
```

在例子中，MASK=0xFC i SHIFT=2.

```
Value1 = 10111000 & 11111100 = 10111000
Key = 10111000 >> 2 = 00101110 -> 0x2E in hexadecimal
```

返回值将对应一个队列规定的内部过滤器句柄(比如说：2:0)。如果这个编号的过滤器存在的话，将核对管制和测量条件(如果过滤器有这些的话)，并返回类编号(我们的例子中是 2:1)，存储于 skb->tc_index 变量。

但是如果找到了那个编号的过滤器，结果将取决于是否声明了 fall_through 标志。如果有，key 值将作为类编号返回。如果没有，将返回一个错误并继续处理其它过滤器。要当心的是如果你使用了 fall_through 标志，并且在 skb->tc_index 和类编号之间存在着一个简单的关系。Be careful if you use fall_through flag; this can be done if a simple relation exists between values of skb->tc_index variable and class id's.

最后要说的参数是 hash 和 pass_on。前者与散列表的大小有关。后者用于指出如

果找不到这个过滤器的返回值指定的类编号，就进行下一个过滤器。缺省行为是 is fall_through (参见下表)。

最后，让我们看一看 TCINDEX 的参数都有那些可能的值：

```
TC Name                 Value          Default
-----------------------------------------------------------------
Hash                    1...0x10000    Implementation dependent
Mask                    0...0xffff     0xffff
Shift                   0...15         0
Fall through / Pass_on  Flag           Fall_through
Classid                 Major:minor    None
Police                  .....          None
```

这种过滤器的功能非常强大，有必要挖掘它所有的可能性。此外，这种过滤器不仅仅适用于 Diffserv 配置，你可以与其它任意类型的过滤器一起使用。

我建议你看看 iproute2 中自带的所有的 Diffserv 范例。我保证我会尽力补全这个文档。此外，我所解释的内容是很多测试的结果。如果你能指出我的错误我将非常感激。

14.4. 入口队列规定

迄今为止所讨论的队列规定都是出口队列规定。每个网卡其实还都可以拥有一个入口队列规定，它不是用来控制向网络上发数据包的，而是用来让你对从网络上收到的数据包应用 tc 过滤器的，而不管它们是发到本机还是需要转发。

由于 tc 过滤器包括了完整地实现了令牌桶过滤器，并且也能够匹配内核的流评估，所以能够实现非常多的功能。这些能够有效地帮助你对输入流进行管制，甚至在它尚未进入 IP 协议层之前。

14.4.1. 参数与使用

输入队列规定本身并不需要任何参数。它不同于其它队列规定之处在于它不占用网卡的根。象这样设置：

```
# tc qdisc add dev eth0 ingress
```

这能够让你除了拥有输入队列规定之外，还有其它的发送队列规定。

关于输入队列的一些人为的例子，请参看 “ 方便菜谱 ”。

14.5. RED(Random Early Detection，随机提前检测)

这一节的意图是引入骨干路由，经常包括<100Mbps 的速率，这同您家里的 ADSL MODEM 和 CABLE MODEM 的实现有所不同。

Internet 上路由器里面的队列的普遍行为称作 “ 截尾 ”。截尾的原理就是排队到一定量之后就开始丢弃后来 “ 溢出 ” 的数据包。这非常的不公平，并且还会引发重

发同步问题，也就是说路由器突然地突发性丢包会导致被丢包者后来突发地重发，进一步地造成路由器拥塞。

为了克服链路上的瞬时拥塞，骨干路由器经常做成大队列。不幸的是，这样做提高了吞吐量的同时也造成了延迟的增加，并导致 TCP 连接在拥挤的时候速率变得非常不均衡。

这些由于截尾带来的问题在 Internet 上不友好的应用程序越来越多而变得越来越棘手。Linux 内核提供了 RED(Random Early Detection，随机提前检测)，更直观的名字叫 Random Early Drop(随机提前丢包)。

RED 并不能解决所有问题，那些未能实现指数 backoff 的应用程序仍然会不公平地共享带宽，但是有了 RED 至少它们不会对其它连接的吞吐量和延迟造成太大的影响。

RED 在数据流没有达到它的带宽硬限制之前就统计地丢弃流中的数据包。让拥塞的骨干链路比较缓和地慢下来，从而防止重传同步问题。因为有意地丢弃了一些数据包，从而可控地减小队列尺寸和延迟，这有助于让 TCP 更快地找到它们 “ 公平的 ” 传输速率。特定连接的包被丢弃的可能性取决于它的带宽占用，而不取决于它发包的数量。

RED 在骨干网上一种很好的队列，尤其是那些无法提供公平队列的场合，因为基于会话状态的分别跟踪很困难。

为了使用 RED，你必须确定三个参数：min、max 和 burst。Min 设置了队列达到多少字节时开始进行丢包，Max 是一个软上限，让算法尽量不要超过，burst 设置了最多有多少个数据包能够突发通过。

对于 min 的值，你应该算出可以接受的最高延迟，然后乘以你的带宽。比如，在我的 64kbps 的 ISDN 上，我希望基础队列延迟是 200ms，所以我设置 min 的值是 1600 字节。min 的值设置得太小会影响吞吐量，而太大则会影响延迟。在慢速的链路上，把 min 设小并不能代替减小 MTU 来改善交互响应。

为了避免重传同步，你至少要把 max 的值设置为 min 的两倍。在慢速链路上，由于 min 的值较小，最好应该设置 max 的值是 min 的四倍以上。

Burst 控制 RED 算法如何响应突发。Burst 必须设置大于 min/avpkt。经验上讲，我发现 (min+min+max)/(3*avpkt)就不错。

另外，你需要设置 limit 和 avpkt。Limit 是一个非常安全的值，当队列中有了这么多字节之后，RED 就开始截尾。我一般设置 limit 值为 max 的八倍。Avpkt 应该是你的平均包大小，在 MTU 是 1500 字节的 Internet 链路上，1000 就可以。

关于技术信息，请阅读 Sally Floyd 和 Van Jacobson 的关于 RED 队列的文章。

14.6. GRED(Generic Random Early Detection,一般的随机提前检测)

关于 GRED 不了解多少。好象 GRED 有很多内置的队列，根据 Diffserv tcindex 的值来决定选择哪个内部队列。根据这里找到的一个幻灯片，它包含了 CISCO 的“分布式加权 RED”和 Dave Clark 的 RIO 的功能。

每个虚拟队列都能分别设置丢包参数。

求助: get Jamal or Werner to tell us more

14.7. VC/ATM 模拟

你可以在 TCP/IP 套接字上建立虚点路，这是 Werner Almesberger 的一个只要成果。虚电路是 ATM 的一个概念。

更多细节请参照 ATM on Linux 网页。

14.8. WRR(Weighted Round Robin，加权轮转)

这个队列规定没有包括在标准内核中，弹你可以从这里下载。迄今，这个队列规定仅在 Linux 2.2 上测试过，但大概也能够在 2.4/2.5 上工作。

WRR 队列规定在使用了 WRR 的各个类之间分配带宽。也就是说，想 CBQ 队列规定那样，它能够包含子类并插入任意队列规定。所有满足要求的类都会按照该类权值的比例分得带宽。权值可以利用 tc 程序手工设置。但对于那些要传输达量数据的类也可以设置为自动递减。

队列规定有一个内置分类器，用来把来自或发往不同机器的数据包分到不同的类。可以使用 MAC 地址/IP 地址和源 IP 地址/目标 IP 地址。MAC 只有在你的 Linux 机器作为一个网桥的时候才能使用。根据数据包的情况，自动把类分配到某一台机器上。

这个队列规定对于象宿舍这样许多不相关的个体共享一个 Internet 连接的时候非常有用。由一套脚本来为 WRR 中心的机器设置相关的行为。

第 15 章 方便菜谱

本章收录了一些“方便菜谱”，希望能够帮你解决实际问题。方便菜谱不能代替理解原理，所以你还是应该着重领会其内在道理。

15.1. 用不同的 SLA 运行多个网站。

你有很多种方法实现这个。Apache 的有些模块可以支持这个功能，但是我们会让你看看 Linux 如何处理这个问题，并能够提供其它多种服务的。这些命令都是从 Jamal Hadi 的演示中偷学来的。

比如说我们有两个顾客，需要 http、ftp 和音频流服务，我们需要向他们出售一定量的带宽。我们在服务器上这么做。

A 顾客应该拥有最多 2Mbps 的带宽，B 顾客则交了 5M 的钱。我们在服务器上分配不同的 IP 地址来区分它们。

```
# ip address add 188.177.166.1 dev eth0
# ip address add 188.177.166.2 dev eth0
```

给服务器赋予那些 IP 地址完全取决于你。当前所有的守护程序都支持这个特性。

我们首先往 eth0 上附加一个 CBQ 队列规定：

```
# tc qdisc add dev eth0 root handle 1: cbq bandwidth 10Mbit cell 8 avpkt 1000 \
  mpu 64
```

为我们的顾客创建类：

```
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 10Mbit rate \
  2MBit avpkt 1000 prio 5 bounded isolated allot 1514 weight 1 maxburst 21
# tc class add dev eth0 parent 1:0 classid 1:2 cbq bandwidth 10Mbit rate \
  5Mbit avpkt 1000 prio 5 bounded isolated allot 1514 weight 1 maxburst 21
```

然后为我们的客户添加过滤器：

```
##求助: Why this line, what does it do?, what is a divisor?:
##求助: A divisor has something to do with a hash table, and the number of
##       buckets - ahu
# tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 1: u32 divisor 1
# tc filter add dev eth0 parent 1:0 prio 5 u32 match ip src 188.177.166.1
  flowid 1:1
# tc filter add dev eth0 parent 1:0 prio 5 u32 match ip src 188.177.166.2
  flowid 1:2
```

做完了。

求助: why no token bucket filter? is there a default pfifo_fast fallback somewhere?

15.2. 防护 SYN 洪水攻击

根据 Alexey 的 iproute 文档和许多非正式的 netfilter 方法而设计。如果你使用这个脚本，请认真调整其中的数值以适应你的网络。

如果你需要保护整个网络，就不要考虑这个脚本了，它最适合于单机使用。

似乎你需要很新版本的 iproute2 工具才能利用 2.4.0 的内核工作。

```
#! /bin/sh -x
#
# sample script on using the ingress capabilities
# this script shows how one can rate limit incoming SYNs
# Useful for TCP-SYN attack protection. You can use
# IPchains to have more powerful additions to the SYN (eg
# in addition the subnet)
#
#path to various utilities;
#change to reflect yours.
#
TC=/sbin/tc
IP=/sbin/ip
IPTABLES=/sbin/iptables
INDEV=eth2
#
# tag all incoming SYN packets through $INDEV as mark value 1
############################################################
$iptables -A PREROUTING -i $INDEV -t mangle -p tcp --syn \
  -j MARK --set-mark 1
############################################################
#
# install the ingress qdisc on the ingress interface
############################################################
$TC qdisc add dev $INDEV handle ffff: ingress
############################################################

#
#
# SYN packets are 40 bytes (320 bits) so three SYNs equals
# 960 bits (approximately 1kbit); so we rate limit below
# the incoming SYNs to 3/sec (not very useful really; but
#serves to show the point - JHS
############################################################
$TC filter add dev $INDEV parent ffff: protocol ip prio 50 handle 1 fw \
police rate 1kbit burst 40 mtu 9k drop flowid :1
############################################################


#
echo "---- qdisc parameters Ingress  ----------"
$TC qdisc ls dev $INDEV
echo "---- Class parameters Ingress  ----------"
$TC class ls dev $INDEV
echo "---- filter parameters Ingress ----------"
$TC filter ls dev $INDEV parent ffff:

#deleting the ingress qdisc
#$TC qdisc del $INDEV ingress
```

15.3. 为防止 DDoS 而对 ICMP 限速

最近一段，分布式拒绝服务攻击成了 Internet 上最让人讨厌的东西。通过对你的网络正确地设置过滤和限速可以避免成为攻击的目标和跳板。

你应该过滤你的网络，禁止非本地 IP 源地址的数据包离开网络，这可以阻止其它人向 Internet 上发送垃圾包。

限速就象以前所展示的那样。如果忘了，就是这张图：

```
[Internet] ---<E3、T3 之类>--- [Linux 路由器] --- [办公室+ISP]
                              eth1        eth0
```

我们先进行预备设置：

```
# tc qdisc add dev eth0 root handle 10: cbq bandwidth 10Mbit avpkt 1000
# tc class add dev eth0 parent 10:0 classid 10:1 cbq bandwidth 10Mbit rate \
  10Mbit allot 1514 prio 5 maxburst 20 avpkt 1000
```

如果你有一个100Mbps的网卡，调整这些数据。现在你需要决定允许多大的ICMP流量。你可以用 tcpdump 测量段时间，把结果写入一个文件，看一看有多少 ICMP 数据包流经网络。别忘了适当延长监测时间！

如果没有条件进行测量，不妨就选择带宽的 5%。设置我们的类：

```
# tc class add dev eth0 parent 10:1 classid 10:100 cbq bandwidth 10Mbit rate \
  100Kbit allot 1514 weight 800Kbit prio 5 maxburst 20 avpkt 250 \
  bounded
```

限速为 100Kbps。然后我们设置过滤器把 ICMP 数据包送给这个类：

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 100 u32 match ip
  protocol 1 0xFF flowid 10:100
```

15.4. 为交互流量设置优先权

当有很多上行和下行数据充斥了你的链路，而你正在使用 telnet 或者 ssh 维护一些系统的话，情况会很糟。器它的数据包会挡住你的按键数据包。有没有一种方法，能让我们的交互数据包在大批数据传输中取得优先呢？Linux 可以实现！

像以前一样，我们得处理量个方向的数据流。明显地，这在链路两端都是 Linux 的时候非常有效——虽然其它的 UNIX 也能实现。问问你周围的 Solaris/BSD 大拿们。

标准的 pfifo_fast 调度器有 3 个频道。0 频道中的数据优先传输，然后才是 1 频道、2 频道。关键就在于让我们的交互数据通过 0 频道！

我们改编自马上就要过期的 ipchains HOWTO：

在 IP 头部有 4 个不常使用的 bit，叫做 TOS 位。它们影响数据包的待遇，4 个 bit 分别代表“最小延迟”、“最大吞吐”、“最大可靠”和“最小成本”。同时只允许设置一位。Rob van Nieuwkerk——ipchains TOS-mangling 代码的作者——这样说：

> “最小延迟”对我来说特别重要。我在一个 33k6 的 MODEM 链路后面。Linux 把数据包区分到 3 个队列中。我在上行(Linux)路由器上为“交互”数据包打开它。这样我就可以在大批下在数据的同时得到可接受的交互性能了。

最通常的用发就是为设置 telnet 和 ftp 控制连接设置“ 最小延迟 ”，并为 ftp 数据连接设置“ 最大吞吐 ”。在你的上行路由器上如此实现：

```
# iptables -A PREROUTING -t mangle -p tcp --sport telnet \
  -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp \
  -j TOS --set-tos Minimize-Delay
# iptables -A PREROUTING -t mangle -p tcp --sport ftp-data \
  -j TOS --set-tos Maximize-Throughput
```

现在，所有从你的 telnet/ftp 主机到达你本机方向的数据流就设置好了。另外一个方向的设置似乎应该由你来设置，比如 telnet、ssh 等等都应自动在发出的数据包上做好 TOS 标记。你随时都可以用 netfilter 来实现。在你的本机上：

```
# iptables -A OUTPUT -t mangle -p tcp --dport telnet \
  -j TOS --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp \
  -j TOS --set-tos Minimize-Delay
# iptables -A OUTPUT -t mangle -p tcp --dport ftp-data \
  -j TOS --set-tos Maximize-Throughput
```

15.5. 使用 netfilter、iproute2 和 squid 实现 WEB 透明代理

本节内容由读者 Ram Narula(泰国)从 Internet 提交用于教学。

在 Linux 上实现这个功能的最正规方法可能是确认数据包的目标端口是 80(web)后，使用 ipchains/iptables 把它们路由到运行着 squid 的服务器上。

有 3 种常规手段来实现，我们这里着重介绍第 4 种。

让网关路由器来路由

> 你可以告诉路由器把目标端口是 80 的数据包发送到 squid 服务器上。
>
> 但是
>
> 这会给你的路由器增加负载，况且很多商业路由器根本就不支持。

使用 4 层交换器

> 4 层交换器支持这个功能毫无问题。
>
> 但是
>
> 这种设备的价格通常非常高。典型 4 层交换器的价格要超过一个典型路由器外加一个好 Linux 服务器的价格。

让 cache 服务器做网关

> 你可以强迫所有的数据包都流经 cache 服务器。

但是

这有相当风险,因为squid会占用很多CPU时间造成总体网络性能的下降，而且服务器本身的崩溃会造成所有人都无法上网。

Linux+NetFilter 路由器

使用 NetFilter 的另一种技术，我们可以让 Netfilter 给去往 80 口的数据包打上一个标记,然后用 iproute2 把打过标记的数据包路由给 Squid 服务器。

```
|----------------|
|      实现      |
|----------------|

 地址使用情况
 10.0.0.1 naret (NetFilter 服务器)
 10.0.0.2 silom (Squid 服务器)
 10.0.0.3 donmuang (连接 Internet 的路由器)
 10.0.0.4 kaosarn (网络上的其它服务器)
 10.0.0.5 RAS 远程访问服务器
 10.0.0.0/24 主网络
 10.0.0.0/19 全体网络

|---------------|
|   网络结构    |
|---------------|

Internet
|
donmuang
|
------------hub/switch----------
|        |             |       |
naret   silom        kaosarn  RAS etc.
```

首先，让 naret 成为除 silom 之外所有工作站的缺省网关，让所有的数据包都流经它。Silom 的缺省网关是 donmuang (10.0.0.3)，否则就会造成 web 数据的环流。

(all servers on my network had 10.0.0.1 as the default gateway which was the former IP address of donmuang router so what I did was changed the IP address of donmuang to 10.0.0.3 and gave naret ip address of 10.0.0.1)

```
Silom
-----
-设置 squid 和 ipchains
```

在 silom 上设置好 squid，保证它能支持透明缓冲/代理，缺省服务端口是 3128，所以所有去网 80 口的数据流都应该重定向到它的 3128 口上。这可以用 ipchains 命令完成：

```
silom# ipchains -N allow1
silom# ipchains -A allow1 -p TCP -s 10.0.0.0/19 -d 0/0 80 -j REDIRECT 3128
silom# ipchains -I input -j allow1
```

或者，用 netfilter 来表达：

```
silom# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 3128
```

(注意：你可能还有其它条目)

关于 squid 服务器的配置，请参考 Squid 常见问题网页：http://squid.nlanr.net)。

请确认这台服务器的 IP 转发已经打开，且缺省网关指向了路由器 donmuang(不是 NOT naret)。

```
Naret
-----
-设置 iptables 和 iproute2
-如果需要，禁止 icmp REDIRECT 消息
```

1. "Mark" 给去往 80 口的数据包打上标记“ 2 ”

```
naret# iptables -A PREROUTING -i eth0 -t mangle -p tcp --dport 80 \
 -j MARK --set-mark 2
```

2. 设置 iproute2，把带有标记“ 2 ”的数据包路由到 silom

```
naret# echo 202 www.out >> /etc/iproute2/rt_tables
naret# ip rule add fwmark 2 table www.out
naret# ip route add default via 10.0.0.2 dev eth0 table www.out
naret# ip route flush cache
```

如果 donmuang 和 naret 在同一个子网中,naret 不应发出 icmp 重定向消息。所以应该禁止 icmp REDIRECT：

```
naret# echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects
naret# echo 0 > /proc/sys/net/ipv4/conf/default/send_redirects
naret# echo 0 > /proc/sys/net/ipv4/conf/eth0/send_redirects
```

设置完成，检查配置：

```
在 naret 上：

naret# iptables -t mangle -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
MARK       tcp  --  anywhere             anywhere           tcp dpt:www MARK set 0x2

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

naret# ip rule ls
0:      from all lookup local
32765:  from all fwmark        2 lookup www.out
32766:  from all lookup main
32767:  from all lookup default

naret# ip route list table www.out
default via 203.114.224.8 dev eth0

naret# ip route
10.0.0.1 dev eth0  scope link
10.0.0.0/24 dev eth0  proto kernel  scope link  src 10.0.0.1
127.0.0.0/8 dev lo  scope link
default via 10.0.0.3 dev eth0
```

```
(make sure silom belongs to one of the above lines, in this case
it's the line with 10.0.0.0/24)

|------|
|-结束-|
|------|
```

15.5.1. 实现之后的数据流图

```
|----------------------------------------|
|Traffic flow diagram after implementation|
|----------------------------------------|

INTERNET
/\
||
\/
-----------------donmuang router---------------------
/\                                     /\         ||
||                                     ||         ||
||                                     \/         ||
naret                                 silom       ||
*destination port 80 traffic==========>(cache)    ||
/\                                     ||         ||
||                                     \/         \/
\\===================================kaosarn, RAS, etc.
```

请注意网络是非对称的，输出的路径多了一跳。

```
Here is run down for packet traversing the network from kaosarn
to and from the Internet.

For web/http traffic:
kaosarn http request->naret->silom->donmuang->internet
http replies from Internet->donmuang->silom->kaosarn

For non-web/http requests(eg. telnet):
kaosarn outgoing data->naret->donmuang->internet
incoming data from Internet->donmuang->kaosarn
```

15.6. 与 PMTU 发现有关的“ 基于路由的 MTU 设置 ”

在传输大量数据的时候，使用大一些的数据包会让 Internet 表现好些。每个数据包都意味着一次路由判断，如果要发送 1M 字节的文件，即使让包处于最大尺寸的话只需要大约 700 个包，而让包尽量小的话则需要 4000 个包。

然而，不是 Internet 的所有部分都能支持每个数据包 1460 字节负载。因此，为了优化连接，就得检测一下，找到数据包最大的合适尺寸。

这个过程就叫“ 沿途 MTU 发现 ”，MTU 意思是“ 最大传输单位 ”。

如果一个路由器发现一个数据包太大而无法在一个数据包中传输，而且数据包标明了“ 不可分片 ”，就会返回一个 ICMP 消息，报告说不得不丢弃这个包。发包的主机根据这个提示就会减小发包尺寸，反复几次之后，就能够找到一个连接沿

特定路径传输时最优的包尺寸。

但是好景不长，因为 Internet 被一帮竭尽全力破坏通信的臭流氓们发现了。然后导致网管们被误导去通过阻挡或限制 ICMP 流量的方法来增加他们 Internet 服务的安全性和坚固性。

现在的情况就是“ 沿途 MTU 发现 ”工作得越来越不好了甚至在某些路由器上失败，以至于 TCP/IP 会话在一段时间之后奇怪地中止。

虽然我不能证明，但是我曾经有两个运行 Alteon Acedirectors 的曾经站点有这个问题，也许有人能提供更多线索说明为什么发生。

15.6.1. 解决方案

当你碰到被这个问题困扰的网站,可以手工地禁止沿途 MTU 发现功能。Koos van den Hout 写到(经简单编辑)：

> 下列问题：因为我的专线只有 33k6，我设置了 ppp 的 MTU 值是 296，但是我无法影响对端的队列。设置为 296 后，击健的响应时间比较合理。
>
> 在我这一端，是一个运行了 Linux 的伪装路由器。
>
> 最近，我把服务器和路由器分开了，所以绝大多数应用程序挪到了路由器的外面。
>
> 然后麻烦了，我无法登录到 irc 里。可怕！经过研究发现我已经连接到了 irc 上，但是没有从 irc 那里收到 motd。我检查了可能出错的地方，注意到以前我的 MTU 是 1500 时，连接某些 web 服务器没有问题，但设置为 296 后不行了，应该是与 MTU 相关的问题。因为 irc 服务器阻挡几乎每种它不直接需要的流量，也包括 icmp。
>
> 我试图向一个 web 服务器的操作员证明这就是引起问题的原因，但他们不愿意解决问题。
>
> 所以我不得不保证输出伪装过的数据包的时候使用比外部链路更低的 MTU。但是我又希望本地以太网使用普通的 MTU(为了 nfs 之类服务)。
>
> 解决方案：

```
ip route add default via 10.0.0.1 mtu 296
```

> (10.0.0.1 是缺省网关，伪装路由器的内网地址)

一般而言，可以通过设置特定的路由器来抑制 PMTU 发现。比如，如果只有某些子网由这个问题，那么：

```
ip route add 195.96.96.0/24 via 10.0.0.1 mtu 1000
```

15.7. 与 PMTU 发现有关的 MSS 箝位（给 ADSL，cable，PPPoE 和 PPtP 用户）

象上面解释的那样，PMTU 发现不再像以前那样有效了。所以如果你知道网络中某一跳的 MTU 比较小(<1500)，你不能指望通过 PMTU 发现来解决。

除了 MTU 之外，还有令一个方式来控制最大包尺寸，所谓的 MSS(Maximum Segment Size，最大段尺寸)。这是 SYN 数据包中的一个 TCP 选项。

最近的 Linux 内核和一些 PPPoE 驱动程序(最显眼的是出色的 Roaring Penguin) 引入了 " MSS 箝位 " 特性。

它的好处在于，设置了 MSS 的值就等于明确地告诉对方 " 不要向我发送大于这个数值的数据包 "。这完全不需要 ICMP 协议的参与。

它的坏处是，它明显是一种 hack——通过改变数据包破坏了 " 端到端 " 原则。知道了这些，我们在很多地方使用这个窍门，向符咒一样灵。

这个功能至少需要 iptables-1.2.1a 和 Linux 2.4.3 才行。基本命令行是：

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS  --clamp-mss-to-pmtu
```

它为你的链路计算出合适的 MSS 值。如果你足够勇敢，或者是高手，你也可以这样做：

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,RST SYN -j TCPMSS --set-mss 128
```

它设置了发送 SYN 的数据包的 MSS 为 128。如果你有小数据包的 VoIP 应用，而巨大的 http 数据包导致了语音的不连续，就可以这样设置。

15.8. 终极的流量控制：低延迟、高速上/下载

Note: This script has recently been upgraded and previously only worked for Linux clients in your network! So you might want to update if you have Windows machines or Macs in your network and noticed that they were not able to download faster while others were uploading.

我试图打造圣杯：

让交互数据包保持较低的延迟时间

> 也就是说上载或下载文件不会打扰 SSH/telnet 等。这是最重要的，即使是 200ms 的延迟也会感觉很差。

上载或下载期间有合理的速率用于网页浏览

> 即使 http 属于一种大量数据传输，也不应受其它传输影响太大。

保证上载不会影响下载

上载数据流会影响下载的速率，这是相当普遍的现象。

只要花费一点点带宽就可以实现它们。之所以上载流、下载流和 ssh 之间要互相伤害，是因为像 Cable 或者 DSL Modem 这样的家用设备中的队列太大。

下面一节进一步解释了造成延迟的原因和如何缓解。如果你对魔术的咒语不感兴趣，完全可以不看它，直接参考脚本那一节。

15.8.1. 为什么缺省设置不让人满意

ISP 们知道人们评价他们的时候要看看下载的速度。除了可用带宽之外，丢包因为会严重地牵制 TCP/IP 效率而极大地影响下载速度。大队列有助于改善丢包，进而提高下载速度。所以 ISP 们都配置成大队列。

然而，大队列会破坏交互性。一次击键行为首先要在上行流队列中排队(可能长达数秒！)才能到达对端主机。回显的时候，数据包还要回来，所以还要在你 ISP 的下行流队列中排队，你的本端显示器才有显示。

这个 HOWTO 教你如何用多种方法重组和处理队列，但是，并不是所有的队列配置我们都有权访问。你的 ISP 的队列肯定是无法配置的。上行流可能存在于你的 cable modem 或 DSL 设备中，你也许可以配置，也许不可以配置。绝大多数不能配置。

那怎么办呢？既然无法控制那些队列，那就除去它们，让数据包在我们的 Linux 路由器内排队。幸运的是，这是可能的。

限制上载速率

> 把上载速率限制在比可用带宽稍小一些的位置上，于是你的 MODEM 中就不会形成队列了。也就是说，队列转移到你的 Linux 路由器中了。

限制下载速率

> 这带点欺骗的味道，因为我们实际上不能控制 Internet 向我们发包的速率。但我们可以丢掉那些太快到来的数据包，不让他们导致 TCP/IP 的速率低于我们期望的速率。因为我们不希望轻易地丢弃数据包，所以我们要配置“ burst ”来容纳突发传输。

在做完了这些之后，我们就排除了下行队列(除了偶尔的突发)，并让上行队列存在于我们的 Linux 路由器中，以便施展 Linux 非凡的能力。

剩下的事情就是保证交互数据包永远排在上行队列的最前面。为了保证上行数据流不会伤害下行流，我们还要把 ACK 数据包排在队列前面。这就是当发生大批量数据流的时候，双向传输均受到严重影响的原因。因为下行数据的 ACK 必须同上行流进行竞争，并在处理过程中被延迟。

为了进一步配置，我们使用一个高质量的 ADSL 连接(荷兰的 xs4all)得出了如下测量结果：

```
基准延迟：
round-trip min/avg/max = 14.4/17.1/21.7 ms
```

```
不进行流量控制，下载数据：
round-trip min/avg/max = 560.9/573.6/586.4 ms

不进行流量控制，上载数据：
round-trip min/avg/max = 2041.4/2332.1/2427.6 ms

进行流量控制，220kbit/s 上行：
round-trip min/avg/max = 15.7/51.8/79.9 ms

进行流量控制，850kbit/s 下行：
round-trip min/avg/max = 20.4/46.9/74.0 ms
```

上载数据时，下载得到约 80%的带宽。当上行流达到 90%时，延迟一下子增加到 850 ms，也能说明问题。

这个脚本的期望效果极大地取决于你的实际上行速率。当你满速率上载的时候，你击键的那个数据包前面总会有一个数据包。这就是你能够达到的上行延迟的下限——MTU 除以上行速率。典型值会比它稍高一些。要达到更好效果，可以降低 MTU！

然后是两个版本的脚本一个是使用 Devik 的 HTB，另一个是使用 Linux 内部自带的 CBQ。都已经过测试，正常工作。

15.8.2. 实际的脚本(CBQ)

适用于所有内核。我们在 CBQ 的内部放置 2 个随机公平队列，以保证多个大吞吐量数据流之间不会互相淹没。

下行流使用带有令牌桶过滤器的 tc 过滤器进行管制。

你可以往脚本中“ tc class add .. classid 1:20 ”那一行添加“ bounded ”进行改进。如果你降低了 MTU，同时也得降低 allot 和 avpkt！

```
#!/bin/bash

# The Ultimate Setup For Your Internet Connection At Home
#
#
# Set the following values to somewhat less than your actual download
# and uplink speed. In kilobits
DOWNLINK=800
UPLINK=220
DEV=ppp0

# clean existing down- and uplink qdiscs, hide errors
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

###### uplink

# install root CBQ

tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth 10mbit

# shape everything at $UPLINK speed - this prevents huge queues in your
```

```
# DSL modem which destroy latency:
# main class

tc class add dev $DEV parent 1: classid 1:1 cbq rate ${UPLINK}kbit \
allot 1500 prio 5 bounded isolated

# high prio class 1:10:

tc class add dev $DEV parent 1:1 classid 1:10 cbq rate ${UPLINK}kbit \
   allot 1600 prio 1 avpkt 1000

# bulk and default class 1:20 - gets slightly less traffic,
#  and a lower priority:

tc class add dev $DEV parent 1:1 classid 1:20 cbq rate $[9*$UPLINK/10]kbit \
   allot 1600 prio 2 avpkt 1000

# both get Stochastic Fairness:
tc qdisc add dev $DEV parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10

# start filters
# TOS Minimum Delay (ssh, NOT scp) in 1:10:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff  flowid 1:10

# ICMP (ip protocol 1) in the interactive class 1:10 so we
# can do measurements & impress our friends:
tc filter add dev $DEV parent 1:0 protocol ip prio 11 u32 \
                                                                m
atch ip protocol 1 0xff flowid 1:10

# To speed up downloads while an upload is going on, put ACK packets in
# the interactive class:

tc filter add dev $DEV parent 1: protocol ip prio 12 u32 \
   match ip protocol 6 0xff \
   match u8 0x05 0x0f at 0 \
   match u16 0x0000 0xffc0 at 2 \
   match u8 0x10 0xff at 33 \
   flowid 1:10

# rest is 'non-interactive' ie 'bulk' and ends up in 1:20

tc filter add dev $DEV parent 1: protocol ip prio 13 u32 \
   match ip dst 0.0.0.0/0 flowid 1:20

########## downlink #############
# slow downloads down to somewhat less than the real speed  to prevent
# queuing at our ISP. Tune to see how high you can set it.
# ISPs tend to have *huge* queues to make sure big downloads are fast
#
# attach ingress policer:

tc qdisc add dev $DEV handle ffff: ingress

# filter *everything* to it (0.0.0.0/0), drop everything that's
# coming in too fast:

tc filter add dev $DEV parent ffff: protocol ip prio 50 u32 match ip src \
   0.0.0.0/0 police rate ${DOWNLINK}kbit burst 10k drop flowid :1
```

如果你希望这个脚本在 ppp 连接时执行，就把它拷贝成/etc/ppp/ip-up.d。

如果在最后两行报错，请更新你的 tc 工具！

15.8.3. 实际的脚本(HTB)

下面的脚本通过 HTB 完成那些功能，参见相关章节。你真的值得为它打个补丁！

```
#!/bin/bash

# The Ultimate Setup For Your Internet Connection At Home
#
#
# Set the following values to somewhat less than your actual download
# and uplink speed. In kilobits
DOWNLINK=800
UPLINK=220
DEV=ppp0

# clean existing down- and uplink qdiscs, hide errors
tc qdisc del dev $DEV root    2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

###### uplink

# install root HTB, point default traffic to 1:20:

tc qdisc add dev $DEV root handle 1: htb default 20

# shape everything at $UPLINK speed - this prevents huge queues in your
# DSL modem which destroy latency:

tc class add dev $DEV parent 1: classid 1:1 htb rate ${UPLINK}kbit burst 6k

# high prio class 1:10:

tc class add dev $DEV parent 1:1 classid 1:10 htb rate ${UPLINK}kbit \
   burst 6k prio 1

# bulk & default class 1:20 - gets slightly less traffic,
# and a lower priority:

tc class add dev $DEV parent 1:1 classid 1:20 htb rate $[9*$UPLINK/10]kbit \
   burst 6k prio 2

# both get Stochastic Fairness:
tc qdisc add dev $DEV parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10

# TOS Minimum Delay (ssh, NOT scp) in 1:10:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff  flowid 1:10

# ICMP (ip protocol 1) in the interactive class 1:10 so we
# can do measurements & impress our friends:
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
                                                                                m
atch ip protocol 1 0xff flowid 1:10

# To speed up downloads while an upload is going on, put ACK packets in
# the interactive class:
```

```
tc filter add dev $DEV parent 1: protocol ip prio 10 u32 \
   match ip protocol 6 0xff \
   match u8 0x05 0x0f at 0 \
   match u16 0x0000 0xffc0 at 2 \
   match u8 0x10 0xff at 33 \
   flowid 1:10

# rest is 'non-interactive' ie 'bulk' and ends up in 1:20


########## downlink #############
# slow downloads down to somewhat less than the real speed  to prevent
# queuing at our ISP. Tune to see how high you can set it.
# ISPs tend to have *huge* queues to make sure big downloads are fast
#
# attach ingress policer:

tc qdisc add dev $DEV handle ffff: ingress

# filter *everything* to it (0.0.0.0/0), drop everything that's
# coming in too fast:

tc filter add dev $DEV parent ffff: protocol ip prio 50 u32 match ip src \
   0.0.0.0/0 police rate ${DOWNLINK}kbit burst 10k drop flowid :1
```

如果你希望这个脚本在 ppp 连接时执行，就把它拷贝成/etc/ppp/ip-up.d。

如果在最后两行报错，请更新你的 tc 工具！

15.9. 为单个主机或子网限速

虽然在我们的 man 和其它地方对于这个问题描述的相当详细，但是这个经常被问到的问题存在简单的回答，不需要完全地理解流量控制。

这招就三行：

```
tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth 10mbit


tc class add dev $DEV parent 1: classid 1:1 cbq rate 512kbit \

allot 1500 prio 5 bounded isolated


tc filter add dev $DEV parent 1: protocol ip prio 16 u32 \

match ip dst 195.96.96.97 flowid 1:1
```

第一行在你的网卡中添加一个基于类的队列，并告诉内核如何计算，假定是一个 10M 的网卡。如果你做错了，不会有什么实际的损害。但是正确地设置会使结果更加精确。

第二行用一些合理的参数创建了一个 512kbit 的类。关于细节，参见 cbq 的手册和第 9 章。

最后一行表明那些数据包应该交给整形的类。未匹配的数据包就不整形。如果想

进行更精确的匹配(子网、端口等等)，参见 9.6.2。

如果你修改了什么，就得重新装入脚本，执行“ tc qdisc del dev $DEV root ”可以清除当前配置。

脚本还可以进一步地改进，最后加上一行“ tc qdisc add dev $DEV parent 1:1 sfq perturb 10 ”。它的作用和细节参见 9.2.3。

15.10. 一个完全 NAT 和 QoS 的范例

我是 Pedro Larroy <piotr@omega.resa.es>

我这里描述的环境是这样的：有很多用户，通过一个 Linux 路由器连接到 Internet 上，路由器上有一个公网 IP 地址，通过 NAT 让大家上网。我作为网管，使用这个 QoS 设置来让我们宿舍的 198 个用户访问 Internet。这里的用户们非常多地使用对等应用，所以合理的带宽管理是必须的。我希望这个例子能够帮助那些对 lartc 感兴趣的读者们。

我一开始一步步地作了实际的实现，最后我会解释如何让这个过程在启动期间自动进行。这个网络的私有 LAN 通过 Linux 路由器连接到 Internet，路由器上有一个公网 IP。把它扩展成拥有多个公网 IP 非常简单，改动一些 iptables 规则即可。为了顺利配置，我们需要：

Linux 2.4.18 或更高版本的内核

如果你使用 2.4.18，需要打上 HTB 补丁。

iproute

也是要确认“ tc ”是否支持 HTB。

iptables

15.10.1. 开始优化那不多的带宽

首先，我们设置一些用来分类的队列规定。我们创建一个由 6 个类组成的 HTB 队列规定，各类按升序排列。这样，我们就有了一些总能够分配到制定带宽的类，但是他们还可以利用其它类没有用到的一部分带宽。更高优先权(更小的优先权值)的类优先获得剩余的带宽。我们的连接是一个 2Mbps 下行/300kbps 上行的 ADSL 线路。我使用 240kbps 作为峰值速率，因为这个值比延迟开始增长时的带宽值稍高一点，延迟的原因是在本端与对端主机之间任意地点的缓冲区造成的。参数应该通过试验计时得到，当观测到附近的主机之间有明显的延迟时就要调高或调低。

现在调整 CEIL 为你上行速率的 75%，我写 eth0 的地方应该是你拥有公网 IP 的网卡。在 root shell 下执行下列命令行，开始我们的试验：

```
CEIL=240
tc qdisc add dev eth0 root handle 1: htb default 15
```

```
tc class add dev eth0 parent 1: classid 1:1 htb rate ${CEIL}kbit ceil ${CEIL}kbit
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 80kbit ceil 80kbit prio 0
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 80kbit ceil ${CEIL}kbit prio 1
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 20kbit ceil ${CEIL}kbit prio 2
tc class add dev eth0 parent 1:1 classid 1:13 htb rate 20kbit ceil ${CEIL}kbit prio 2
tc class add dev eth0 parent 1:1 classid 1:14 htb rate 10kbit ceil ${CEIL}kbit prio 3
tc class add dev eth0 parent 1:1 classid 1:15 htb rate 30kbit ceil ${CEIL}kbit prio 3
tc qdisc add dev eth0 parent 1:12 handle 120: sfq perturb 10
tc qdisc add dev eth0 parent 1:13 handle 130: sfq perturb 10
tc qdisc add dev eth0 parent 1:14 handle 140: sfq perturb 10
tc qdisc add dev eth0 parent 1:15 handle 150: sfq perturb 10
```

我们刚才创建了一个一级的 HTB 树。看上去应该是：

```
+---------+
| root 1: |
+---------+
     |
+---------------------------------------+
| class 1:1                             |
+---------------------------------------+
  |      |      |      |      |      |
+----+ +----+ +----+ +----+ +----+ +----+
|1:10| |1:11| |1:12| |1:13| |1:14| |1:15|
+----+ +----+ +----+ +----+ +----+ +----+
```

classid 1:10 htb rate 80kbit ceil 80kbit prio 0

> 这是一个较高优先权的类。这个类中的数据包拥有最低的延迟并最先取得空闲带宽，所以应该设置这个类的峰值速率。我们将把这些要求低延迟的服务归属到该类中：*ssh*、*telnet*、*dns*、*quake3*、*irc 和带有 SYN 标记的数据包。*

classid 1:11 htb rate 80kbit ceil ${CEIL}kbit prio 1

> 这是我们第一个用于放置大批量传输的类。在我的例子中，用来处理从我的本地 WEB 服务器发出和浏览网页的数据包，分别是源端口 80、目标端口 80 的包。

classid 1:12 htb rate 20kbit ceil ${CEIL}kbit prio 2

> 我把带有最大吞吐 TOS 位的数据包和其余从路由器*本地进程*发往 Internet 的数据包放在这个类中。所以下面的类会包括经过路由穿过机器的数据包。

classid 1:13 htb rate 20kbit ceil ${CEIL}kbit prio 2

> 这些类包含了其它经过 NAT，需要用高优先权进行大批量传输的机器。

classid 1:14 htb rate 10kbit ceil ${CEIL}kbit prio 3

> 这里是邮件(SMTP、pop3…)相关和 TOS 要求最小成本的数据流。

classid 1:15 htb rate 30kbit ceil ${CEIL}kbit prio 3

> 最后是路由器后面经过 NAT 进行大批量传输的机器。所有的变态网虫、

下载狂人等等都分在这里，以保证他们不会妨碍正常服务。

15.10.2. 对数据包分类

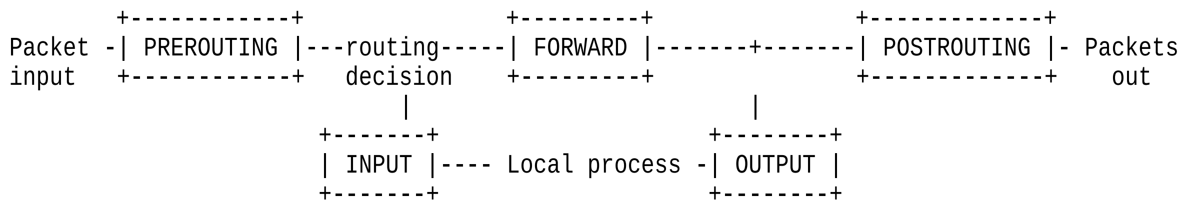
我们创建好了队列规定，但是还没有进行必要的分类，所以现在等于把所有发出的数据包都送给了 1:15 (因为我们设的是：tc qdisc add dev eth0 root handle 1: htb *default 15*)。现在需要告诉机器那些数据包走哪条路，这是最重要的部分。

现在我们设置过滤器以便用 iptables 对数据包进行分类。我确实更愿意用 iptables 来做这件事，因为它更灵活，而且你还可以为每个规则设置计数器。而且可以通过 RETURN 方法避免遍历所有的规则。我们执行下面的命令：

```
tc filter add dev eth0 parent 1:0 protocol ip prio 1 handle 1 fw classid 1:10
tc filter add dev eth0 parent 1:0 protocol ip prio 2 handle 2 fw classid 1:11
tc filter add dev eth0 parent 1:0 protocol ip prio 3 handle 3 fw classid 1:12
tc filter add dev eth0 parent 1:0 protocol ip prio 4 handle 4 fw classid 1:13
tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 5 fw classid 1:14
tc filter add dev eth0 parent 1:0 protocol ip prio 6 handle 6 fw classid 1:15
```

这样我们就告诉了内核，数据包会有一个特定的 FWMARK 标记值(hanlde x fw)，表明它应该送给哪个类(classid x:x)。后面你会明白如何给数据包打标记。

你必须先搞明白数据包穿越 Linux 的 IP 协议层：



我假定你已经完整地清除了 tables 而且缺省策略是 ACCEPT(-P ACCEPT)。如果你还没有搞明白 iptables 也没关系，缺省设置也可以用。我们的私有子网是一个 B 类——172.17.0.0/16，我们的公网 IP 是 212.170.21.172。

下面我们指示内核进行 NAT，以便私网内部的人能够与外部通讯：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -s 172.17.0.0/255.255.0.0 -o eth0 -j SNAT --to-source
212.170.21.172
```

然后检查数据包是否通过了 1:15：

```
tc -s class show dev eth0
```

你可以开始给数据包打标记了，往 PREROUTING 链中添加一个 mangle 规则：

```
iptables -t mangle -A PREROUTING -p icmp -j MARK --set-mark 0x1
iptables -t mangle -A PREROUTING -p icmp -j RETURN
```

现在你从私网内部 ping 外部的某个站点，就会看到数据包的计数在增长。察看

1:10 的计数：

```
tc -s class show dev eth0
```

我们设置了一个-j RETURN 以防止数据包遍历所有的规则。也就是说 icmp 数据包不必再匹配下面的那些规则了。把这招记住。现在我们可以开始添加更多规则了，我们来彻底地设置 TOS 的处理：

```
iptables -t mangle -A PREROUTING -m tos --tos Minimize-Delay -j MARK --set-mark 0x1
iptables -t mangle -A PREROUTING -m tos --tos Minimize-Delay -j RETURN
iptables -t mangle -A PREROUTING -m tos --tos Minimize-Cost -j MARK --set-mark 0x5
iptables -t mangle -A PREROUTING -m tos --tos Minimize-Cost -j RETURN
iptables -t mangle -A PREROUTING -m tos --tos Maximize-Throughput -j MARK --set-mark 0x6
iptables -t mangle -A PREROUTING -m tos --tos Maximize-Throughput -j RETURN
```

然后提高 ssh 数据包的优先权：

```
iptables -t mangle -A PREROUTING -p tcp -m tcp --sport 22 -j MARK --set-mark 0x1
iptables -t mangle -A PREROUTING -p tcp -m tcp --sport 22 -j RETURN
```

提高 tcp 初始连接(也就是带有 SYN 的数据包)的优先权是非常明智的：

```
iptables -t mangle -I PREROUTING -p tcp -m tcp --tcp-flags SYN,RST,ACK SYN -j MARK --set-mark
0x1
iptables -t mangle -I PREROUTING -p tcp -m tcp --tcp-flags SYN,RST,ACK SYN -j RETURN
```

以此类推。当我们向 PREROUTING 中添加完 mangle 规则后，用这条规则结束 PREROUTING 表：

```
iptables -t mangle -A PREROUTING -j MARK --set-mark 0x6
```

也就是说前面没有打过标记的数据包将交给 1:15 处理。实际上最后一步是不必要的，因为 1:15 是缺省类，但我仍然打上标记是为了保持整个设置的协调一致，而且这样还能看到规则的包计数。

应该在 OUTPUT 链中再重复一遍上面的设置，也就是说把命令中的 –A PREROUTING 改成-A OUTPUT (s/PREROUTING/OUTPUT/)。那么本地(Linux 路由器)产生的数据包也可以参与分类了。我用-j MARK --set-mark 0x3 来结束 OUTPUT 链，让本地产生的数据包有更高的优先权。

15.10.3. 改进设置

现在我们完成了所有的工作。花点时间看看图表，了解一下你的带宽究竟在被如何消耗，与你的期望是否一致。经过很多小时的监测，最终让 Internet 连接达到了很好的效果。否则可能发生经常性地超时或者 tcp 的新建连接无法得到任何带宽。

如果你发现有些类几乎总是满满的，不如在那个类下面再附加上另一个队列规定，以保证带宽的公平使用：

```
tc qdisc add dev eth0 parent 1:13 handle 130: sfq perturb 10
```

```
tc qdisc add dev eth0 parent 1:14 handle 140: sfq perturb 10
tc qdisc add dev eth0 parent 1:15 handle 150: sfq perturb 10
```

15.10.4. 让上面的设置开机时自动执行

肯定有无数种方法。我的方法是，写一个脚本/etc/init.d/packetfilter，让他能够接受[start | stop | stop-tables | start-tables | reload-tables]等参数，来配置对列规定和装载需要的内核模块，就像一个守护程序一样。同样的脚本从/etc/network/iptables-rules 中得到 iptables 规则。我争取稍稍美化一下，放到我的网页上去。

第 16 章 构建网桥以及用 ARP 代理构

网桥是一种安装在网络中，不需要任何后续配置的设备。网络交换器基本上就是一个多口网桥。也就是说网桥就是一个两口的交换器。而 Linux 能支持多个接口的网桥，成为一个真正的交换器。

网桥经常被用于改进那些工作状态不佳但是又不能改造的网络。因为网桥是一个 2 层设备（IP 下面的那一层），路由器和服务器意识不到它的存在。也就意味着你可以完全透明地阻挡或者修改数据包，甚至流量整形。

另一件好事是，假如一个网桥崩溃了，可以使用一个 HUB 甚至一根交叉线来代替它。

坏消息是，如果它没有在工程文档中明确备案，网桥就会导致混乱。它不会在 traceroute 中出现，但是却导致数据包从 A 点到 B 点过程中消失或者变样(“ 网上闹鬼 !”)。你应该也想知道一个 “ 不想改变任何事 ” 的组织是否在做应该做的事情。

Linux 2.4/2.5 网桥的文档在这个网页里.

16.1. 桥接与 iptables 的关系

截止到 Linux 2.4.20，桥接和 iptables 在不借助其他条件的时候无法互相看到。如果你把数据包从 eth0 桥接到 eth1，它不会通过 iptables。这意味着你无法进行包过滤、NAT、mangle 等等操作。从 Linux 2.5.45 开始，这个问题已被解决。

你也可能听说过另一个叫做“ ebtables ”计划，它可以实现 MACNAT 和“ brouting ”等等疯狂的功能。这确实令人振奋！

16.2. 桥接与流量整形

做个广告：没问题！

只是要明确哪块网卡在哪一边，否则你可能会在内部网卡上配置外网流量整形，那当然不能工作。必要的话使用嗅探器确认。

16.3. 用 ARP 代理实现伪网桥

如果你只是想实现一个伪网桥，请直接阅读“ 实现 ”一节，但是看看它是如何工作的不是坏事。

伪网桥的工作有些特别。缺省情况下，网桥不加改变地把数据帧从一个端口发送到另一个端口。它只是看看数据帧的硬件地址来决定这个帧应该送到哪里去。也就是说，只要数据帧有合适的硬件地址，你可以让 Linux 转发它并不认识的数据帧。

而伪网桥的工作有些区别，它看上去更像是一个隐形的路由器，而不是一个网桥。但类似于网桥的是，对网络的设计没有太大影响。

因为它不是一个网桥，就具有了一个优势：数据帧（包）会通过内核，所以你可以进行过滤、修改、重定向或者重路由。

一个真的网桥也可以实现上述技巧，但那需要特定的代码，象以太帧分流器或者上面提到的那个补丁。

伪网桥的另一个好处是它不会转发它不认识的数据包，这样可以防止一些 cruft 充斥网络，从而净化你的网络环境。如果你确实需要这些 cruft（比如 SAP 包或者 NETBEUI），就应该使用真网桥。

16.3.1. ARP 和 ARP 代理

当一个主机希望向同一物理网段上的另一台主机发包时，它会发送一个 ARP 包询问：“ 请问 10.0.0.7，你的硬件地址是多少？告诉 10.0.0.1 ”。为了将来回应，10.0.0.1 会告诉对方自己的硬件地址。

然后 10.0.0.7 向发出询问的机器发出回应。得到别人的硬件地址后，机器会 cache 住较长的一段时间，cache 过期后，它会再次询问。

如果想建立一个伪网桥，我们就应该指示我们的网桥，让它回应这些 ARP 询问，导致这个网络上的机器都把包发给我们的伪网桥。我们的伪网桥会处理这些数据包，把它们发往相关的网卡。

简单地说，就是当一端的主机询问另一端主机的硬件地址的时候，伪网桥做出回应“ 把数据包交给我 ”。

这样一来，所有的数据包都能被发送到正确的地方，并且总是通过网桥。

16.3.2. 实现

在万恶的旧社会，只能指示内核对任意子网进行 ARP 代理。所以，要想配置一个伪网桥，你就得正确设置两个网卡间的双向路由，并添加匹配代理 ARP 的规则。这是很繁琐的，需要敲进很多命令，所以容易导致错误，造成你的伪网桥回应了某个子网的 ARP 请求，但它却不知道如何处理。

使用了 Linux 2.4/2.5(可能也有 2.2),这种局限被打破了，可以使用/proc 目录下的一个称为“ proxy_arp ”的标记来操控。建立伪网桥的过程就变成了：

1. 为一左一右两块网卡分配 IP 地址
2. 配置路由好让你的机器知道那些主机在左边，哪些主机在右边
3. 开启两块网卡的 ARP 代理

```
echo 1 > /proc/sys/net/ipv4/conf/ethL/proxy_arp
echo 1 > /proc/sys/net/ipv4/conf/ethR/proxy_arp
```

L 和 R 就是你左右两边网卡的编号。

不要忘了打开 ip_forwarding 标记！因为桥接并不需要这个，所以当你把一个真网桥转换成一个伪网桥的时候可能发现这个标记是关闭的。

你在转换过程中还可能预见量一个问题：网络中主机的 arp 缓冲中仍然保存着设置伪网桥之前的错误硬件地址。

在 Cisco 上，用命令 “ `clear arp-cache` ” 就可以，在 Linux 上，使用 “ arp -d ip 地址 ” 来实现。你也可以等待缓冲过期，这可能需要相当长的时间。

你可以使用 “ arping ” 工具来加速这个过程，很多发行版都把它作为 “ iputils ” 的一部分进行发布。你可以是用 “ arping ” 来主动发送 ARP 信息，来更新对方的 ARP 缓冲。

这是一种非常强有力的技术，“ black hats ” 就是用这种技术来搅乱你的路由的！

- 如果想在 Linux 2.4 上主动发送 ARP 消息，你就得先执行：

  ```
  echo 1 > /proc/sys/net/ipv4/ip_nonlocal_bind
  ```

假如在这之前，你在写路由的时候有忽略掩码这种毛病的话，你还会发现网络的配置出现了错误。因为有些版本的 route 命令当时猜测了你的掩码，猜错了你也不知道。所以，如果做了上述那种特外科手术般的路由设置，一定要检查你的掩码设置是否正确！

第 17 章 动态路由——OSPF 和 BGP

当你的网络变得确实比较大，或者你开始考虑成为国际互联网的一部分的时候，你就需要能够动态调整路由的工具了。站点之间经常由越来越多个链路互相连接。

OSPF 和 BGP4 几乎已经成了 Internet 的实际标准。通过 gated 和 zebra，Linux 全都能支持。

因为这些内容暂时没有包括进本文档，我们在此仅给出具体的学习方向：

纵览：

Cisco 系统设计大规模的 IP 网络

对于 OSPF：

Moy, John T. "OSPF，Internet 路由协议剖析" Addison Wesley. Reading, MA. 1998.

Halabi 也写了一个 OSPF 路由设计方面很好的指南，但是好象这篇文章已经从 Cisco 的主页上砍掉了。

对于 BGP：

Halabi, Bassam "Internet 的路由体系" Cisco 出品(New Riders 出版社). 印第安纳波利斯, 1997.

另外，

Cisco 系统

使用边界网关协议进行域间路由

虽然上面的例子都是针对 Cisco 的，但是它与 Zebra 说使用的配置语言明显很相似☺。

17.1. 用 Zebra 设置 OSPF

如果下列信息有不准确的地方或者你有任何建议，请尽快通知我。Zebra 是由 Kunihiro Ishiguro、Toshiaki Takada 和 Yasuhiro Ohara 编写的一个非常出色的动态路由软件。利用 Zebra，设置 OSPF 非常快捷，但是在实践中，如果你由特别的要求，就必须调整很多参数。OSPF 的意思是 Open Shortest Path First(开放式最短路径优先)，它的主要特点是：

分层

网络被分成*区域*，各区域通过被指定为 0 号区域的*骨干区域*进行互连，0

区域内的所有路由器都知道其它每个区域的路由信息。

收敛快

相对于 RIP 来说，路由的传播非常快。

省带宽

使用了多播来代替广播，所以不会打扰对路由信息不感兴趣的主机，同时减少了网络传输。同时，*内部路由器*不需知道其它区域的路由信息。那些用不同接口把不同区域连在一起的路由器叫做*边界路由器*，它们保存邻接区域的拓扑信息。

CPU 要求高

OSPF 是基于 Dijkstra 的 [最短路径优先算法]的，该算法比其它的路由算法需要更复杂的计算。但幸运的是，只有在区域内部才进行最短路径运算，而且对于中小规模的网络而言并不是问题，甚至你根本感觉不到。

链路状态

OSPF 要考虑网络和接口的特别信息，比如带宽、链路异常和成本。

协议开放而且软件是 GPL 的

OSPF 是一个开放协议，而且 Zebra 是一个 GPL 软件，项对于专有的软件和协议而言有这明显的优势。

17.1.1. 必要条件

Linux 内核：

编译时要加上 CONFIG_NETLINK_DEV 和 CONFIG_IP_MULTICAST 的支持。(我不大确定是否还需要别的).

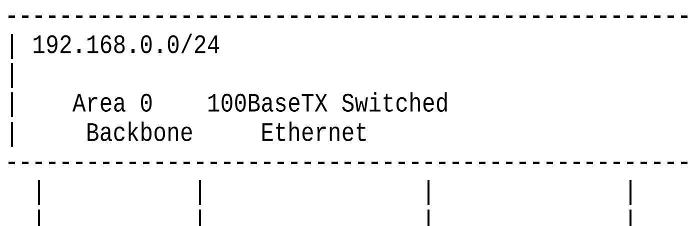
iproute

Zebra

从 http://www.zebra.org 下载，或者找一个适合你系统的打包软件。

17.1.2. 配置 Zebra

让我们用这个网络作为例子：



```
    |eth1       |eth1            |eth0          |
    |100BaseTX  |100BaseTX       |100BaseTX     |100BaseTX
    |.1         |.2              |.253          |
   ---------   ------------   -----------      ----------------
   |R Omega|   |R Atlantis|   |R Legolas|      |R Frodo       |
   ---------   ------------   -----------      ----------------
    |eth0         |eth0              |             |          |
    |             |                  |             |          |
    |2MbDSL/ATM   |100BaseTX         |10BaseT      |10BaseT   |10BaseT
  ------------------------------------------------------------------------------------------
  | Internet |   | 172.17.0.0/16       Area 1      |       |  192.168.1.0/24 wlan  Area 2|
  ------------   |        Student network (dorm)    |       |      barcelonawireless      |
                 ------------------------------------       -------------------------------
```

不要被这张图吓住，zebra 会自动完成绝大多数事情，也就是说 zebra 会把所有的路由添加好，不需要我们做什么具体工作。用手工的方式花几天时间维护路由表是很痛苦的事情。最重要的事情就是要明确网络的拓扑结构。要特别注意区域 0，因为它最重要。先来配置 zebra，修改 zebra.conf 适应你的需求：

```
hostname omega
password xxx
enable password xxx
!
! Interface's description.
!
!interface lo
! description test of desc.
!
interface eth1
multicast
!
! Static default route
!
ip route 0.0.0.0/0 212.170.21.129
!
log file /var/log/zebra/zebra.log
```

在 Debian 环境里，还要修改/etc/zebra/daemons 以保证开机的时候能被启动：

```
zebra=yes
ospfd=yes
```

然后修改 ospfd.conf（如果使用 IPv6 的话就要修改 ospf6d.conf）。我的 ospfd.conf 的内容是：

```
hostname omega
password xxx
enable password xxx
!
router ospf
  network 192.168.0.0/24 area 0
  network 172.17.0.0/16 area 1
!
! log stdout
log file /var/log/zebra/ospfd.log
```

告诉了 ospf 我们网络的拓扑结构。

17.1.3. 运行 Zebra

现在，我们来运行 Zebra，可以直接敲入“ zebra -d ”命令，也可以使用诸如“ /etc/init.d/zebra start ”之类的脚本。然后注意看 ospdfd 的日志，应该看到类似这样的内容：

```
2002/12/13 22:46:24 OSPF: interface 192.168.0.1 join AllSPFRouters Multicast group.
2002/12/13 22:46:34 OSPF: SMUX_CLOSE with reason: 5
2002/12/13 22:46:44 OSPF: SMUX_CLOSE with reason: 5
2002/12/13 22:46:54 OSPF: SMUX_CLOSE with reason: 5
2002/12/13 22:47:04 OSPF: SMUX_CLOSE with reason: 5
2002/12/13 22:47:04 OSPF: DR-Election[1st]: Backup 192.168.0.1
2002/12/13 22:47:04 OSPF: DR-Election[1st]: DR     192.168.0.1
2002/12/13 22:47:04 OSPF: DR-Election[2nd]: Backup 0.0.0.0
2002/12/13 22:47:04 OSPF: DR-Election[2nd]: DR     192.168.0.1
2002/12/13 22:47:04 OSPF: interface 192.168.0.1 join AllDRouters Multicast group.
2002/12/13 22:47:06 OSPF: DR-Election[1st]: Backup 192.168.0.2
2002/12/13 22:47:06 OSPF: DR-Election[1st]: DR     192.168.0.1
2002/12/13 22:47:06 OSPF: Packet[DD]: Negotiation done (Slave).
2002/12/13 22:47:06 OSPF: nsm_change_status(): scheduling new router-LSA origination
2002/12/13 22:47:11 OSPF: ospf_intra_add_router: Start
```

现在先忽略 SMUX_CLOSE 消息，因为那是有关 SNMP 的。我们能看到 192.168.0.1 是*指定的路由器*，而 192.168.0.2 是*后备指定路由器*。

我们也可以用下述命令直接控制 zebra 或 ospfd：

```
$ telnet localhost zebra
$ telnet localhost ospfd
```

让我们看一看路由是否正在扩散，登录到 zebra 后敲入：

```
root@atlantis:~# telnet localhost zebra
Trying 127.0.0.1...
Connected to atlantis.
Escape character is '^]'.
                                                                        H
ello, this is zebra (version 0.92a).
Copyright 1996-2001 Kunihiro Ishiguro.
                                                                        U
ser Access Verification
Password:
atlantis> show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       B - BGP, > - selected route, * - FIB route

K>* 0.0.0.0/0 via 192.168.0.1, eth1
C>* 127.0.0.0/8 is directly connected, lo
O   172.17.0.0/16 [110/10] is directly connected, eth0, 06:21:53
C>* 172.17.0.0/16 is directly connected, eth0
O   192.168.0.0/24 [110/10] is directly connected, eth1, 06:21:53
C>* 192.168.0.0/24 is directly connected, eth1
atlantis> show ip ospf border-routers
============ OSPF router routing table =============
R    192.168.0.253         [10] area: (0.0.0.0), ABR
via 192.168.0.253, eth1
[10] area: (0.0.0.1), ABR
via 172.17.0.2, eth0
```

或者直接用 iproute 察看：

```
root@omega:~# ip route
212.170.21.128/26 dev eth0  proto kernel  scope link  src 212.170.21.172
192.168.0.0/24 dev eth1  proto kernel  scope link  src 192.168.0.1
172.17.0.0/16 via 192.168.0.2 dev eth1  proto zebra  metric 20
default via 212.170.21.129 dev eth0  proto zebra
root@omega:~#
```

我们能看到以前并不存在的 zebra 路由。在启动了 zebra 和 ospfd 之后仅仅几秒钟就能够看到这些路由，很有成就感。你可以 ping 一下其它主机以便检验一下连通性。Zebra 的路由配置是自动的，你可以向网络里再添加一台路由器，配置好 zebra，然后喊 “ 乌拉 !”

提示：你可以使用：

```
t
cpdump -i eth1 ip[9] == 89
```

来抓几个 OSPF 分析一下。OSPF 的 IP 协议号码是 89，而协议声明字段在 IP 头的第 9 个字节处。

OSPF 有很多可调整的参数，特别在大网络环境下。将来这篇 HOWTO 进一步扩展的时候，我们将讲解一些微调 OSPF 的方法。

第 18 章 其它可能性

这章列出了一些有关 Linux 的高级路由和流量整形的计划。其中有些链接应该单独写一章，有些本身的文档十分完整，不需要更多的 HOWTO。

802.1Q VLAN 在 Linux 上的实现(网站)

VLAN 是一种使用非物理方法把网络划分成多个部分的技术。这里可以找到很多有关 VLAN 的信息。利用这个实现，你可以让你的 Linux 机器与 VLAN 设备(比如 Cisco Catalyst，3Com: <Corebuilder, Netbuilder II, SuperStack II switch 630>，Extreme Ntwks Summit 48，Foundry: <ServerIronXL, FastIron>)交流。

这里有关于 VLAN 非常好的 HOWTO。

更新：这个计划已经归入了 2.4.14 (也可能是 13)版的内核。

802.1Q VLAN 在 Linux 上的另一个实现(网站)

如题。这个计划从一开始与已经确立的 vlan 计划的体系和编码风格并无不协调之处，结果可以有一个非常干净的整体设计。

Linux 虚拟服务器(网站)

这些人很聪明。Linux 虚拟服务器是由一个 Linux 系统的负载均衡器把多个真实服务器组织起来，成为一个高度可伸缩、高可靠服务器方案。对于最终用户而言，集群的内部结构识完全透明的，他们只能看到一个单一的虚拟服务器。

简单地说，无论你针对哪一层流量和是否需要负载均衡，LVS 总有方法来实现。他们的一些技术非常绝妙！比如，让同一网段上的几台机器使用相同的 IP，但是关闭它们的 ARP。只有 LVS 服务器才使用 ARP——由它来决定哪个后端服务器来处理到来的数据包，然后把它发送给对应后端服务器的 MAC 地址。出去的数据包则直接发给路由器，而不必经过 LVS 机器，也就是 LVS 不会看到你那发到全世界的 5Gbps 数据流，从而避免了成为瓶颈。

LVS 在 Linux2.0 和 2.2 上以内核补丁的方式实现，而在 2.4/2.5 上则以 Netfilter 模块的方式实现，不需要内核补丁。他们的 2.4 支持尚处在早期开发，希望大家捧场并给出回馈和补丁。

CBQ.init (site)

配置 CBQ 确实令人厌烦，尤其当你仅仅是想把路由器后面的几台机器进行流量整形的时候。CBQ.init 可以帮助你用很简单的语法来配置你的 Linux 路由器。

比如，你想把 192.168.1.0/24 子网(连接在 10Mbps 的 eth1)的下载速率限制在 28kbps 上，只要把这些行放在 CBQ init 配置文件中即可：

```
DEVICE=eth1,10Mbit,1Mbit
RATE=28Kbit
WEIGHT=2Kbit
PRIO=5
RULE=192.168.1.0/24
```

如果你对于“ 如何？为什么？ ”这类问题不感兴趣的话，一定得用它。我们在产品中使用了 CBQ init，情况良好。他还可以做一些高级配置，比如整形时间表。文档嵌入在脚本中，所以你找不到单独的 README。

Chronox easy shaping scripts (site)

Stephan Mueller (smueller@chronox.de) 写了两个有用的脚本，“ limit.conn ”和 “ shaper ”。第一个让你轻松地限制一个下载会话，象这样：

```
# limit.conn -s SERVERIP -p SERVERPORT -l LIMIT
```

工作于 Linux 2.2 和 2.4/2.5。

第二个脚本稍稍复杂些，可用于在 iptables 规则上生成很多不同的队列(利用标记和整形)。

虚拟路由器冗余协议的实现(网站)

这纯粹是为了冗余。两台机器，用他们各自的 IP 地址和 MAC 地址构造出第三个虚拟的 IP 地址和 MAC 地址。最初纯粹为需要固定 MAC 地址的路由器而设计，其实也可以用在服务器上。

它的可爱之处在于配置异常简单。不需要补丁和编译内核，全是用户级的。

在共同提供一个服务的服务器上运行：

```
# vrrpd -i eth0 -v 50 10.0.0.22
```

就可以用了！10.0.0.22 现在代表你的一台服务器，可能是运行 vrrp 守护程序的第一台机器。现在把这台机器从网络上断开，令一台机器会迅速接管 10.0.0.22 这个地址及其 MAC 地址。

我这里做了一个试验，并持续运行了 1 分钟。不知为什么，总是 drop my 缺省网关，但是可以用-n 选项避免。

这就是失效恢复现场：

```
64 bytes from 10.0.0.22: icmp_seq=3 ttl=255 time=0.2 ms
64 bytes from 10.0.0.22: icmp_seq=4 ttl=255 time=0.2 ms
64 bytes from 10.0.0.22: icmp_seq=5 ttl=255 time=16.8 ms
64 bytes from 10.0.0.22: icmp_seq=6 ttl=255 time=1.8 ms
64 bytes from 10.0.0.22: icmp_seq=7 ttl=255 time=1.7 ms
```

一个 ping 包也没有丢！在第 4 个包之后我从网络上断开了我的 P200，然后我的 486 来接管，你可以看出延迟提高了。

第 19 章 进一步学习

http://snafu.freedom.org/linux2.2/iproute-notes.html

有很多来自内核的技术信息和注解。

http://www.davin.ottawa.on.ca/ols/

Linux 流量控制的作者之一——Jamal Hadi Salim 的幻灯片。

http://defiant.coinet.com/iproute2/ip-cref/

Alexey 的 LaTeX 文档的 HTML 版本——iproute2 的部分详细讲解。

http://www.aciri.org/floyd/cbq.html

Sally Floyd 有一个关于 CBQ 的非常好的网页，包括她的文件原稿。不仅仅是针对 CBQ，而是精彩地讨论了使用 CBQ 的理论问题。非常专业化的东西，但是对于爱好者来说非常易读。

Differentiated Services on Linux

由 Werner Almesberger、Jamal Hadi Salim 和 Alexey Kuznetsov 所写的这篇文档描述了 Linux 内核的 TBF、GRED、DSMARK 队列规定和 tcindex 分类器中的 DiffServ 机制。

http://ceti.pl/~kravietz/cbq/NET4_tc.html

另一个 HOWTO，不过是波兰语的！但你还是可以从中 copy/paste 命令行，它们与语言无关。作者正在与我们合作，可能很快会为这个 HOWTO 写出一些章节。

IOS Committed Access Rate

由来自 Cisco 的一些乐于助人的家伙，他们有个好习惯——把他们的文档发不到网上。Cisco 命令语法虽然不一样，但是概念一样，除了——我们可以不必花一辆汽车的钱去实现那些功能。:-)

Docum 试验场(这里)

Stef Coene 正在说服他的老板出售 Linux 支持，所以他做了很多试验，特别是关于带宽管理的。他的站点有很多经验、范例、测试并指出了 CBQ/tc 的一些 BUG。

TCP/IP Illustrated, volume 1, W. Richard Stevens, ISBN 0-201-63346-9

如果你真的想理解 TCP/IP 就一定要读一读，很有趣。

第 20 章 鸣谢

我们希望能够列出所有那些对这个 HOWTO 做出贡献或者帮助我们揭示了内在机制的人们。因为目前我们没有计划做一个 Netfilter 类型的计分板，所以我们暂时列出这些帮助了我们的人们。

- Junk Alins <juanjo@mat.upc.es>
- Joe Van Andel
- Michael T. Babcock <mbabcock@fibrespeed.net>
- Christopher Barton <cpbarton%uiuc.edu>
- Ard van Breemen <ard%kwaak.net>
- Ron Brinker <service%emcis.com>
- ?ukasz Bromirski <l.bromirski@mr0vka.eu.org>
- Lennert Buytenhek <buytenh@gnu.org>
- Esteve Camps <esteve@hades.udg.es>
- Stef Coene <stef.coene@docum.org>
- Don Cohen <don-lartc%isis.cs3-inc.com>
- Jonathan Corbet <lwn%lwn.net>
- Gerry N5JXS Creager <gerry%cs.tamu.edu>
- Marco Davids <marco@sara.nl>
- Jonathan Day <jd9812@my-deja.com>
- Martin aka devik Devera <devik@cdi.cz>
- Hannes Ebner <he%fli4l.de>
- Derek Fawcus <dfawcus%cisco.com>
- Stephan "Kobold" Gehring <Stephan.Gehring@bechtle.de>
- Jacek Glinkowski <jglinkow%hns.com>
- Andrea Glorioso <sama%perchetopi.org>
- Nadeem Hasan <nhasan@usa.net>
- Erik Hensema <erik%hensema.xs4all.nl>

- Vik Heyndrickx <vik.heyndrickx@edchq.com>
- Spauldo Da Hippie <spauldo%usa.net>
- Koos van den Hout <koos@kzdoos.xs4all.nl>
- Stefan Huelbrock <shuelbrock%datasystems.de>
- Alexander W. Janssen <yalla%ynfonatic.de>
- Gareth John <gdjohn%zepler.org>
- Dave Johnson <dj@www.uk.linux.org>
- Martin Josefsson <gandalf%wlug.westbo.se>
- Andi Kleen <ak%suse.de>
- Andreas J. Koenig <andreas.koenig%anima.de>
- Pawel Krawczyk <kravietz%alfa.ceti.pl>
- Amit Kucheria <amitk@ittc.ku.edu>
- Edmund Lau <edlau%ucf.ics.uci.edu>
- Philippe Latu <philippe.latu%linux-france.org>
- Arthur van Leeuwen <arthurvl%sci.kun.nl>
- Jose Luis Domingo Lopez <jdomingo@24x7linux.com>
- Jason Lunz <j@cc.gatech.edu>
- Stuart Lynne <sl@fireplug.net>
- Alexey Mahotkin <alexm@formulabez.ru>
- Predrag Malicevic <pmalic@ieee.org>
- Patrick McHardy <kaber@trash.net>
- Andreas Mohr <andi%lisas.de>
- James Morris <jmorris@intercode.com.au>
- Andrew Morton <akpm%zip.com.au>
- Wim van der Most
- Stephan Mueller <smueller@chronox.de>
- Togan Muftuoglu <toganm%yahoo.com>
- Chris Murray <cmurray@stargate.ca>
- Patrick Nagelschmidt <dto%gmx.net>
- Ram Narula <ram@princess1.net>

- Jorge Novo <jnovo@educanet.net>
- Patrik <ph@kurd.nu>
- P?l Osgy?ny <oplab%westel900.net>
- Lutz Preßler <Lutz.Pressler%SerNet.DE>
- Jason Pyeron <jason%pyeron.com>
- Rusty Russell <rusty%rustcorp.com.au>
- Mihai RUSU <dizzy%roedu.net>
- Jamal Hadi Salim <hadi%cyberus.ca>
- Ren? Serral <rserral%ac.upc.es>
- David Sauer <davids%penguin.cz>
- Sheharyar Suleman Shaikh <sss23@drexel.edu>
- Stewart Shields <MourningBlade%bigfoot.com>
- Nick Silberstein <nhsilber%yahoo.com>
- Konrads Smelkov <konrads@interbaltika.com>
- William Stearns <wstearns@pobox.com>
- Andreas Steinmetz <ast%domdv.de>
- Jason Tackaberry <tack@linux.com>
- Charles Tassell <ctassell%isn.net>
- Glen Turner <glen.turner%aarnet.edu.au>
- Tea Sponsor: Eric Veldhuyzen <eric%terra.nu>
- Song Wang <wsong@ece.uci.edu>
- Chris Wilson <chris@netservers.co.uk>
- Lazar Yanackiev <Lyanackiev%gmx.net>
- Pedro Larroy <piotr%omega.resa.es>
    - 第 15 章，第 10 节：Example of a full nat solution with QoS
    - 第 17 章，第 1 节：用 Zebra 设置 OSPF